

Formalization of an Optimized Monitoring Algorithm for Metric First-Order Dynamic Logic with Aggregations

Thibault Dardinier Lukas Heimes Martin Raszyk Joshua Schneider
Dmitriy Traytel

June 14, 2026

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 dynamic logic (MFODL), which combines the features of metric first-order temporal logic (MFOTL) [2] and metric dynamic logic [3]. Thus, MFODL supports real-time constraints, first-order parameters, and regular expressions. Additionally, the monitor supports aggregation operations such as count and sum. This formalization, which is described in a paper at IJCAR 2020 [1], significantly extends [previous work on a verified monitor](#) for MFOTL [4]. Apart from the addition of regular expressions and aggregations, we implemented [multi-way joins](#) and a specialized sliding window algorithm to further optimize the monitor.

Contents

1	Code adaptation for IEEE double-precision floats	2
1.1	copysign	2
1.2	Additional lemmas about generic floats	2
1.3	Doubles with a unified NaN value	4
1.4	Linear ordering	6
1.4.1	Code setup	8
2	Event parameters	9
3	Regular expressions	11
4	Metric first-order dynamic logic	14
4.1	Formulas and satisfiability	15
4.1.1	Syntax	15
4.1.2	Future reach	18
4.1.3	Semantics	18
4.2	Past-only formulas	20
4.3	Safe formulas	20
4.4	Slicing traces	22
4.5	Translation to n-ary conjunction	23
5	Optimized relational join	25
5.1	Binary join	25
5.2	Multi-way join	26

6	Generic monitoring algorithm	30
6.1	Monitorable formulas	30
6.2	Handling regular expressions	31
6.2.1	LPD	33
6.2.2	RPD	35
6.3	The executable monitor	36
6.4	Verdict delay	45
6.5	Specification	48
6.6	Correctness	49
6.6.1	Invariants	49
6.6.2	Initialisation	54
6.6.3	Evaluation	55
6.6.4	Monitor step	70
6.6.5	Monitor function	71
6.7	Collected correctness results	72
7	Efficient implementation of temporal operators	73
7.1	Optimized queue data structure	73
7.2	Optimized data structure for Since	76
7.3	Optimized data structure for Until	83
8	Instantiation of the generic algorithm and code setup	88

1 Code adaptation for IEEE double-precision floats

1.1 copysign

lift_definition *copysign* :: ('e, 'f) float \Rightarrow ('e, 'f) float \Rightarrow ('e, 'f) float is
 $\lambda(_, e::'e \text{ word}, f::'f \text{ word}) (s::1 \text{ word}, _, _). (s, e, f) \langle \text{proof} \rangle$

lemma *is_nan_copysign[simp]*: *is_nan* (*copysign* *x y*) \longleftrightarrow *is_nan* *x*
 $\langle \text{proof} \rangle$

1.2 Additional lemmas about generic floats

lemma *is_nan_some_nan[simp]*: *is_nan* (*some_nan* :: ('e, 'f) float)
 $\langle \text{proof} \rangle$

lemma *not_is_nan_0[simp]*: \neg *is_nan* 0
 $\langle \text{proof} \rangle$

lemma *not_is_nan_1[simp]*: \neg *is_nan* 1
 $\langle \text{proof} \rangle$

lemma *is_nan_plus*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* + *y*)
 $\langle \text{proof} \rangle$

lemma *is_nan_minus*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* - *y*)
 $\langle \text{proof} \rangle$

lemma *is_nan_times*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* * *y*)
 $\langle \text{proof} \rangle$

lemma *is_nan_divide*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* / *y*)
 $\langle \text{proof} \rangle$

lemma *is_nan_float_sqrt*: $is_nan\ x \implies is_nan\ (float_sqrt\ x)$
(*proof*)

lemma *nan_fcompare*: $is_nan\ x \vee is_nan\ y \implies fcompare\ x\ y = Und$
(*proof*)

lemma *nan_not_le*: $is_nan\ x \vee is_nan\ y \implies \neg x \leq y$
(*proof*)

lemma *nan_not_less*: $is_nan\ x \vee is_nan\ y \implies \neg x < y$
(*proof*)

lemma *nan_not_zero*: $is_nan\ x \implies \neg is_zero\ x$
(*proof*)

lemma *nan_not_infinity*: $is_nan\ x \implies \neg is_infinity\ x$
(*proof*)

lemma *zero_not_infinity*: $is_zero\ x \implies \neg is_infinity\ x$
(*proof*)

lemma *zero_not_nan*: $is_zero\ x \implies \neg is_nan\ x$
(*proof*)

lemma *minus_one_power_one_word*: $(-1 :: real) ^{unat\ (x :: 1\ word)} = (if\ unat\ x = 0\ then\ 1\ else\ -1)$
(*proof*)

definition *valofn* :: ('e, 'f) float \Rightarrow real **where**
 $valofn\ x = (2^{exponent\ x} / 2^{bias\ TYPE((\prime e, \prime f)\ float)}) * (1 + real\ (fraction\ x) / 2^{LENGTH(\prime f)})$

definition *valofd* :: ('e, 'f) float \Rightarrow real **where**
 $valofd\ x = (2 / 2^{bias\ TYPE((\prime e, \prime f)\ float)}) * (real\ (fraction\ x) / 2^{LENGTH(\prime f)})$

lemma *valof_alt*: $valof\ x = (if\ exponent\ x = 0\ then\ if\ sign\ x = 0\ then\ valofd\ x\ else\ -\ valofd\ x\ else\ if\ sign\ x = 0\ then\ valofn\ x\ else\ -\ valofn\ x)$
(*proof*)

lemma *fraction_less_2p*: $fraction\ (x :: (\prime e, \prime f)\ float) < 2^{LENGTH(\prime f)}$
(*proof*)

lemma *valofn_ge_0*: $0 \leq valofn\ x$
(*proof*)

lemma *valofn_ge_2p*: $2^{exponent\ (x :: (\prime e, \prime f)\ float)} / 2^{bias\ TYPE((\prime e, \prime f)\ float)} \leq valofn\ x$
(*proof*)

lemma *valofn_less_2p*:
fixes $x :: (\prime e, \prime f)\ float$
assumes $exponent\ x < e$
shows $valofn\ x < 2^e / 2^{bias\ TYPE((\prime e, \prime f)\ float)}$
(*proof*)

lemma *valofd_ge_0*: $0 \leq valofd\ x$
(*proof*)

lemma *valofd_less_2p*: *valofd* (*x* :: ('e, 'f) float) < 2 / 2^{bias} TYPE(('e, 'f) float)
<proof>

lemma *valofn_le_imp_exponent_le*:
fixes *x y* :: ('e, 'f) float
assumes *valofn x* ≤ *valofn y*
shows *exponent x* ≤ *exponent y*
<proof>

lemma *valofn_eq*:
fixes *x y* :: ('e, 'f) float
assumes *valofn x* = *valofn y*
shows *exponent x* = *exponent y* *fraction x* = *fraction y*
<proof>

lemma *valofd_eq*:
fixes *x y* :: ('e, 'f) float
assumes *valofd x* = *valofd y*
shows *fraction x* = *fraction y*
<proof>

lemma *is_zero_valof_conv*: *is_zero x* ↔ *valof x* = 0
<proof>

lemma *valofd_neq_valofn*:
fixes *x y* :: ('e, 'f) float
assumes *exponent y* ≠ 0
shows *valofd x* ≠ *valofn y* *valofn y* ≠ *valofd x*
<proof>

lemma *sign_gt_0_conv*: 0 < *sign x* ↔ *sign x* = 1
<proof>

lemma *valof_eq*:
assumes ¬ *is_zero x* ∨ ¬ *is_zero y*
shows *valof x* = *valof y* ↔ *x* = *y*
<proof>

lemma *zero_fcompare*: *is_zero x* ⇒ *is_zero y* ⇒ *fcompare x y* = *ccode.Eq*
<proof>

1.3 Doubles with a unified NaN value

quotient_type *double* = (11, 52) float / λ*x y*. *is_nan x* ∧ *is_nan y* ∨ *x* = *y*
<proof>

instantiation *double* :: {zero, one, plus, minus, uminus, times, ord}
begin

lift_definition *zero_double* :: *double* **is** 0 <proof>

lift_definition *one_double* :: *double* **is** 1 <proof>

lift_definition *plus_double* :: *double* ⇒ *double* ⇒ *double* **is** plus
<proof>

lift_definition *minus_double* :: *double* ⇒ *double* ⇒ *double* **is** minus
<proof>

```

lift_definition uminus_double :: double  $\Rightarrow$  double is uminus
  <proof>

lift_definition times_double :: double  $\Rightarrow$  double  $\Rightarrow$  double is times
  <proof>

lift_definition less_eq_double :: double  $\Rightarrow$  double  $\Rightarrow$  bool is ( $\leq$ )
  <proof>

lift_definition less_double :: double  $\Rightarrow$  double  $\Rightarrow$  bool is ( $<$ )
  <proof>

instance <proof>

end

instantiation double :: inverse
begin

lift_definition divide_double :: double  $\Rightarrow$  double  $\Rightarrow$  double is divide
  <proof>

definition inverse_double :: double  $\Rightarrow$  double where
  inverse_double x = 1 div x

instance <proof>

end

lift_definition sqrt_double :: double  $\Rightarrow$  double is float_sqrt
  <proof>

no_notation plus_infinity ( $\langle\infty\rangle$ )

lift_definition infinity :: double is plus_infinity <proof>

lift_definition nan :: double is some_nan <proof>

lift_definition is_zero :: double  $\Rightarrow$  bool is IEEE.is_zero
  <proof>

lift_definition is_infinite :: double  $\Rightarrow$  bool is IEEE.is_infinity
  <proof>

lift_definition is_nan :: double  $\Rightarrow$  bool is IEEE.is_nan
  <proof>

lemma is_nan_conv: is_nan x  $\longleftrightarrow$  x = nan
  <proof>

lift_definition copysign_double :: double  $\Rightarrow$  double  $\Rightarrow$  double is
   $\lambda x y.$  if IEEE.is_nan y then some_nan else copysign x y
  <proof>

Note: copysign_double deviates from the IEEE standard in cases where the second argument is a NaN.

lift_definition fcompare_double :: double  $\Rightarrow$  double  $\Rightarrow$  ccode is fcompare
  <proof>

```

lemma *nan_fcompare_double*: $is_nan\ x \vee is_nan\ y \implies fcompare_double\ x\ y = Und$
 ⟨proof⟩

consts *compare_double* :: $double \Rightarrow double \Rightarrow integer$

specification (*compare_double*)

compare_double_less: $compare_double\ x\ y < 0 \iff is_nan\ x \wedge \neg is_nan\ y \vee fcompare_double\ x\ y = ccode.Lt$

compare_double_eq: $compare_double\ x\ y = 0 \iff is_nan\ x \wedge is_nan\ y \vee fcompare_double\ x\ y = ccode.Eq$

compare_double_greater: $compare_double\ x\ y > 0 \iff \neg is_nan\ x \wedge is_nan\ y \vee fcompare_double\ x\ y = ccode.Gt$
 ⟨proof⟩

lemmas *compare_double_simps* = *compare_double_less compare_double_eq compare_double_greater*

lemma *compare_double_le_0*: $compare_double\ x\ y \leq 0 \iff is_nan\ x \vee fcompare_double\ x\ y \in \{ccode.Eq, ccode.Lt\}$
 ⟨proof⟩

lift_definition *double_of_integer* :: $integer \Rightarrow double$ **is**
 $\lambda x. zerosign\ 0\ (intround\ RNE\ (int_of_integer\ x))$ ⟨proof⟩

definition *double_of_int where* $double_of_int\ x = double_of_integer\ (integer_of_int\ x)$

lemma [*code*]: $double_of_int\ (int_of_integer\ x) = double_of_integer\ x$
 ⟨proof⟩

lift_definition *integer_of_double* :: $double \Rightarrow integer$ **is**
 $\lambda x. if\ IEEE.is_nan\ x \vee IEEE.is_infinity\ x\ then\ undefined$
 $else\ integer_of_int\ [valof\ (intround\ roundTowardZero\ (valof\ x)) :: (11, 52)\ float]$
 ⟨proof⟩

definition *int_of_double*: $int_of_double\ x = int_of_integer\ (integer_of_double\ x)$

1.4 Linear ordering

definition *lcompare_double* :: $double \Rightarrow double \Rightarrow integer$ **where**
 $lcompare_double\ x\ y = (if\ is_zero\ x \wedge is_zero\ y\ then$
 $compare_double\ (copysign_double\ 1\ x)\ (copysign_double\ 1\ y)$
 $else\ compare_double\ x\ y)$

lemma *fcompare_double_swap*: $fcompare_double\ x\ y = ccode.Gt \iff fcompare_double\ y\ x = ccode.Lt$
 ⟨proof⟩

lemma *fcompare_double_refl*: $\neg is_nan\ x \implies fcompare_double\ x\ x = ccode.Eq$
 ⟨proof⟩

lemma *fcompare_double_Eq1*: $fcompare_double\ x\ y = ccode.Eq \implies fcompare_double\ y\ z = c \implies fcompare_double\ x\ z = c$
 ⟨proof⟩

lemma *fcompare_double_Eq2*: $fcompare_double\ y\ z = ccode.Eq \implies fcompare_double\ x\ y = c \implies fcompare_double\ x\ z = c$
 ⟨proof⟩

lemma *fcompare_double_Lt_trans*: $fcompare_double\ x\ y = ccode.Lt \implies fcompare_double\ y\ z = ccode.Lt$

$\implies \text{fcompare_double } x \ z = \text{ccode.Lt}$
(proof)

lemma $\text{fcompare_double_eq}$: $\neg \text{is_zero } x \vee \neg \text{is_zero } y \implies \text{fcompare_double } x \ y = \text{ccode.Eq} \implies x = y$
(proof)

lemma $\text{fcompare_double_Lt_asym}$: $\text{fcompare_double } x \ y = \text{ccode.Lt} \implies \text{fcompare_double } y \ x = \text{ccode.Lt}$
 $\implies \text{False}$
(proof)

lemma $\text{compare_double_swap}$: $0 < \text{compare_double } x \ y \longleftrightarrow \text{compare_double } y \ x < 0$
(proof)

lemma $\text{compare_double_refl}$: $\text{compare_double } x \ x = 0$
(proof)

lemma $\text{compare_double_trans}$: $\text{compare_double } x \ y \leq 0 \implies \text{compare_double } y \ z \leq 0 \implies \text{compare_double } x \ z \leq 0$
(proof)

lemma $\text{compare_double_antisym}$: $\text{compare_double } x \ y \leq 0 \implies \text{compare_double } y \ x \leq 0 \implies \neg \text{is_zero } x \vee \neg \text{is_zero } y \implies x = y$
(proof)

lemma $\text{zero_compare_double_copysign}$: $\text{compare_double } (\text{copysign_double } 1 \ x) (\text{copysign_double } 1 \ y) \leq 0 \implies \text{is_zero } x \implies \text{is_zero } y \implies \text{compare_double } x \ y \leq 0$
(proof)

lemma $\text{is_zero_double_cases}$: $\text{is_zero } x \implies (x = 0 \implies P) \implies (x = -0 \implies P) \implies P$
(proof)

lemma $\text{copysign_1_0[simp]}$: $\text{copysign_double } 1 \ 0 = 1$ $\text{copysign_double } 1 \ (-0) = -1$
(proof)

lemma $\text{is_zero_uminus_double[simp]}$: $\text{is_zero } (-x) \longleftrightarrow \text{is_zero } x$
(proof)

lemma $\text{not_is_zero_one_double[simp]}$: $\neg \text{is_zero } 1$
(proof)

lemma $\text{uminus_one_neq_one_double[simp]}$: $-1 \neq (1 :: \text{double})$
(proof)

definition $\text{lle_double} :: \text{double} \Rightarrow \text{double} \Rightarrow \text{bool}$ **where**
 $\text{lle_double } x \ y \longleftrightarrow \text{lcompare_double } x \ y \leq 0$

definition $\text{lless_double} :: \text{double} \Rightarrow \text{double} \Rightarrow \text{bool}$ **where**
 $\text{lless_double } x \ y \longleftrightarrow \text{lcompare_double } x \ y < 0$

lemma $\text{lcompare_double_ge_0}$: $\text{lcompare_double } x \ y \geq 0 \longleftrightarrow \text{lle_double } y \ x$
(proof)

lemma $\text{lcompare_double_gt_0}$: $\text{lcompare_double } x \ y > 0 \longleftrightarrow \text{lless_double } y \ x$
(proof)

lemma $\text{lcompare_double_eq_0}$: $\text{lcompare_double } x \ y = 0 \longleftrightarrow x = y$
(proof)

```

lemmas lcompare_double_0_folds = lle_double_def[symmetric] lless_double_def[symmetric]
lcompare_double_ge_0 lcompare_double_gt_0 lcompare_double_eq_0

```

```

interpretation double_linorder: linorder lle_double lless_double
⟨proof⟩

```

```

instantiation double :: equal
begin

```

```

definition equal_double :: double ⇒ double ⇒ bool where
equal_double x y ↔ lcompare_double x y = 0

```

```

instance ⟨proof⟩

```

```

end

```

```

derive (eq) ceq double

```

```

definition comparator_double :: double comparator where
comparator_double x y = (let c = lcompare_double x y in
  if c = 0 then order.Eq else if c < 0 then order.Lt else order.Gt)

```

```

lemma comparator_double: comparator comparator_double
⟨proof⟩

```

```

⟨ML⟩

```

```

derive ccompare double

```

1.4.1 Code setup

```

declare [[code drop:
  0 :: double
  1 :: double
  plus :: double ⇒ _
  minus :: double ⇒ _
  uminus :: double ⇒ _
  times :: double ⇒ _
  less_eq :: double ⇒ _
  less :: double ⇒ _
  divide :: double ⇒ _
  sqrt_double infinity nan is_zero is_infinite is_nan copysign_double fcompare_double
  double_of_integer integer_of_double
]]

```

code_printing

```

code_module FloatUtil → (OCaml)
⟨module FloatUtil : sig
  val iszero : float -> bool
  val isinfinite : float -> bool
  val isnan : float -> bool
  val copysign : float -> float -> float
  val compare : float -> float -> Z.t
end = struct
  let iszero x = (Pervasives.classify_float x = Pervasives.FP_zero);;
  let isinfinite x = (Pervasives.classify_float x = Pervasives.FP_infinite);;
  let isnan x = (Pervasives.classify_float x = Pervasives.FP_nan);;
  let copysign x y = if isnan y then Pervasives.nan else Pervasives.copysign x y;;

```

```

  let compare x y = Z.of_int (Pervasives.compare x y);;
end;;>

```

```
code_reserved (OCaml) Pervasives FloatUtil
```

```
code_printing
```

```

type_constructor double → (OCaml) float
| constant uminus :: double ⇒ double → (OCaml) Pervasives.(~-. )
| constant (+) :: double ⇒ double ⇒ double → (OCaml) Pervasives.(+. )
| constant (*) :: double ⇒ double ⇒ double → (OCaml) Pervasives.( *. )
| constant (/) :: double ⇒ double ⇒ double → (OCaml) Pervasives.( / . )
| constant (-) :: double ⇒ double ⇒ double → (OCaml) Pervasives.( -. )
| constant 0 :: double → (OCaml) 0.0
| constant 1 :: double → (OCaml) 1.0
| constant (≤) :: double ⇒ double ⇒ bool → (OCaml) Pervasives.( <= )
| constant (<) :: double ⇒ double ⇒ bool → (OCaml) Pervasives.( < )
| constant sqrt_double :: double ⇒ double → (OCaml) Pervasives.sqrt
| constant infinity :: double → (OCaml) Pervasives.infinity
| constant nan :: double → (OCaml) Pervasives.nan
| constant is_zero :: double ⇒ bool → (OCaml) FloatUtil.iszero
| constant is_infinite :: double ⇒ bool → (OCaml) FloatUtil.isinfinite
| constant is_nan :: double ⇒ bool → (OCaml) FloatUtil.isnan
| constant copysign_double :: double ⇒ double ⇒ double → (OCaml) FloatUtil.copysign
| constant compare_double :: double ⇒ double ⇒ integer → (OCaml) FloatUtil.compare
| constant double_of_integer :: integer ⇒ double → (OCaml) Z.to'_float
| constant integer_of_double :: double ⇒ integer → (OCaml) Z.of'_float

```

```
hide_const (open) fcompare_double
```

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 | EFloat double | EString String.literal
```

```
derive (eq) ceq event_data
```

```
derive ccompare event_data
```

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

```
begin
```

```
fun less_eq_event_data where
```

```

  EInt x ≤ EInt y ↔ x ≤ y
| EInt x ≤ EFloat y ↔ double_of_integer x ≤ y
| EInt _ ≤ EString _ ↔ False
| EFloat x ≤ EInt y ↔ x ≤ double_of_integer y
| EFloat x ≤ EFloat y ↔ x ≤ y

```

```

| EFloat _ < EString _ <-> False
| EString x < EString y <-> lexordp_eq (String.explode x) (String.explode y)
| EString _ <= _ <-> False

```

definition *less_event_data* :: *event_data* \Rightarrow *event_data* \Rightarrow *bool* **where**
less_event_data *x y* <-> $x \leq y \wedge \neg y \leq x$

fun *plus_event_data* **where**
EInt *x* + *EInt* *y* = *EInt* (*x* + *y*)
| *EInt* *x* + *EFloat* *y* = *EFloat* (*double_of_integer* *x* + *y*)
| *EFloat* *x* + *EInt* *y* = *EFloat* (*x* + *double_of_integer* *y*)
| *EFloat* *x* + *EFloat* *y* = *EFloat* (*x* + *y*)
| (*_*::*event_data*) + *_* = *EFloat* *nan*

fun *minus_event_data* **where**
EInt *x* - *EInt* *y* = *EInt* (*x* - *y*)
| *EInt* *x* - *EFloat* *y* = *EFloat* (*double_of_integer* *x* - *y*)
| *EFloat* *x* - *EInt* *y* = *EFloat* (*x* - *double_of_integer* *y*)
| *EFloat* *x* - *EFloat* *y* = *EFloat* (*x* - *y*)
| (*_*::*event_data*) - *_* = *EFloat* *nan*

fun *uminus_event_data* **where**
- *EInt* *x* = *EInt* (- *x*)
| - *EFloat* *x* = *EFloat* (- *x*)
| - (*_*::*event_data*) = *EFloat* *nan*

fun *times_event_data* **where**
EInt *x* * *EInt* *y* = *EInt* (*x* * *y*)
| *EInt* *x* * *EFloat* *y* = *EFloat* (*double_of_integer* *x* * *y*)
| *EFloat* *x* * *EInt* *y* = *EFloat* (*x* * *double_of_integer* *y*)
| *EFloat* *x* * *EFloat* *y* = *EFloat* (*x* * *y*)
| (*_*::*event_data*) * *_* = *EFloat* *nan*

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

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

instance <*proof*>

end

primrec *integer_of_event_data* :: *event_data* \Rightarrow *integer* **where**
integer_of_event_data (*EInt* *x*) = *x*
| *integer_of_event_data* (*EFloat* *x*) = *integer_of_double* *x*
| *integer_of_event_data* (*EString* *_*) = 0

primrec *double_of_event_data* :: *event_data* \Rightarrow *double* **where**
double_of_event_data (*EInt* *x*) = *double_of_integer* *x*
| *double_of_event_data* (*EFloat* *x*) = *x*
| *double_of_event_data* (*EString* *_*) = *nan*

3 Regular expressions

context begin

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

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

definition *Wild* = *Skip 1*

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

lemma *size_regex_estimation'*[*termination_simp*]: $x \in \text{atms } r \implies y \leq f x \implies y \leq \text{size_regex } f r$
 ⟨*proof*⟩ **definition** *TimesL* *r S* = *Times r ' S*

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

qualified primrec *fv_regex* **where**

fv_regex fv (Skip n) = {}
 | *fv_regex fv (Test φ)* = *fv φ*
 | *fv_regex fv (Plus r s)* = *fv_regex fv r* \cup *fv_regex fv s*
 | *fv_regex fv (Times r s)* = *fv_regex fv r* \cup *fv_regex fv s*
 | *fv_regex fv (Star r)* = *fv_regex fv r*

lemma *fv_regex_cong*[*fundef_cong*]:
 $r = r' \implies (\bigwedge z. z \in \text{atms } r \implies \text{fv } z = \text{fv}' z) \implies \text{fv_regex } fv r = \text{fv_regex } fv' r'$
 ⟨*proof*⟩

lemma *finite_fv_regex*[*simp*]: ($\bigwedge z. z \in \text{atms } r \implies \text{finite } (fv z)$) $\implies \text{finite } (fv_regex fv r)$
 ⟨*proof*⟩

lemma *fv_regex_commute*:
 $(\bigwedge z. z \in \text{atms } r \implies x \in fv z \iff g x \in fv' z) \implies x \in fv_regex fv r \iff g x \in fv_regex fv' r$
 ⟨*proof*⟩

lemma *fv_regex_alt*: $fv_regex fv r = (\bigcup z \in \text{atms } r. fv z)$
 ⟨*proof*⟩ **definition** *nfv_regex* **where**
nfv_regex fv r = *Max (insert 0 (Suc ' fv_regex fv r))*

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

lemma *nfv_regex_simps*[*simp*]:
assumes [*simp*]: ($\bigwedge z. z \in \text{atms } r \implies \text{finite } (fv z)$) ($\bigwedge z. z \in \text{atms } s \implies \text{finite } (fv z)$)
shows
nfv_regex fv (Skip n) = 0
nfv_regex fv (Test φ) = *Max (insert 0 (Suc ' fv φ))*
nfv_regex fv (Plus r s) = *max (nfv_regex fv r) (nfv_regex fv s)*
nfv_regex fv (Times r s) = *max (nfv_regex fv r) (nfv_regex fv s)*
nfv_regex fv (Star r) = *nfv_regex fv r*
 ⟨*proof*⟩

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

qualified primrec *match* :: (*nat* \Rightarrow 'a \Rightarrow *bool*) \Rightarrow 'a *regex* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool* **where**
match test (Skip n) = ($\lambda i j. j = i + n$)
 | *match test (Test φ)* = ($\lambda i j. i = j \wedge \text{test } i \varphi$)

| $\text{match test } (\text{Plus } r \ s) = \text{match test } r \sqcup \text{match test } s$
| $\text{match test } (\text{Times } r \ s) = \text{match test } r \text{ OO } \text{match test } s$
| $\text{match test } (\text{Star } r) = (\text{match test } r)^{**}$

lemma $\text{match_cong}[\text{fundef_cong}]$:

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

<proof> **primrec** eps **where**

$\text{eps test } i \ (\text{Skip } n) = (n = 0)$

$\text{eps test } i \ (\text{Test } \varphi) = \text{test } i \ \varphi$

$\text{eps test } i \ (\text{Plus } r \ s) = (\text{eps test } i \ r \vee \text{eps test } i \ s)$

$\text{eps test } i \ (\text{Times } r \ s) = (\text{eps test } i \ r \wedge \text{eps test } i \ s)$

$\text{eps test } i \ (\text{Star } r) = \text{True}$

qualified primrec lpd **where**

$\text{lpd test } i \ (\text{Skip } n) = (\text{case } n \ \text{of } 0 \Rightarrow \{\} \mid \text{Suc } m \Rightarrow \{\text{Skip } m\})$

$\text{lpd test } i \ (\text{Test } \varphi) = \{\}$

$\text{lpd test } i \ (\text{Plus } r \ s) = (\text{lpd test } i \ r \cup \text{lpd test } i \ s)$

$\text{lpd test } i \ (\text{Times } r \ s) = \text{TimesR } (\text{lpd test } i \ r) \ s \cup (\text{if } \text{eps test } i \ r \ \text{then } \text{lpd test } i \ s \ \text{else } \{\})$

$\text{lpd test } i \ (\text{Star } r) = \text{TimesR } (\text{lpd test } i \ r) \ (\text{Star } r)$

qualified primrec $\text{lpd}\kappa$ **where**

$\text{lpd}\kappa \ \kappa \ \text{test } i \ (\text{Skip } n) = (\text{case } n \ \text{of } 0 \Rightarrow \{\} \mid \text{Suc } m \Rightarrow \{\kappa \ (\text{Skip } m)\})$

$\text{lpd}\kappa \ \kappa \ \text{test } i \ (\text{Test } \varphi) = \{\}$

$\text{lpd}\kappa \ \kappa \ \text{test } i \ (\text{Plus } r \ s) = \text{lpd}\kappa \ \kappa \ \text{test } i \ r \cup \text{lpd}\kappa \ \kappa \ \text{test } i \ s$

$\text{lpd}\kappa \ \kappa \ \text{test } i \ (\text{Times } r \ s) = \text{lpd}\kappa \ (\lambda t. \kappa \ (\text{Times } t \ s)) \ \text{test } i \ r \cup (\text{if } \text{eps test } i \ r \ \text{then } \text{lpd}\kappa \ \kappa \ \text{test } i \ s \ \text{else } \{\})$

$\text{lpd}\kappa \ \kappa \ \text{test } i \ (\text{Star } r) = \text{lpd}\kappa \ (\lambda t. \kappa \ (\text{Times } t \ (\text{Star } r))) \ \text{test } i \ r$

qualified primrec rpd **where**

$\text{rpd test } i \ (\text{Skip } n) = (\text{case } n \ \text{of } 0 \Rightarrow \{\} \mid \text{Suc } m \Rightarrow \{\text{Skip } m\})$

$\text{rpd test } i \ (\text{Test } \varphi) = \{\}$

$\text{rpd test } i \ (\text{Plus } r \ s) = (\text{rpd test } i \ r \cup \text{rpd test } i \ s)$

$\text{rpd test } i \ (\text{Times } r \ s) = \text{TimesL } r \ (\text{rpd test } i \ s) \cup (\text{if } \text{eps test } i \ s \ \text{then } \text{rpd test } i \ r \ \text{else } \{\})$

$\text{rpd test } i \ (\text{Star } r) = \text{TimesL } (\text{Star } r) \ (\text{rpd test } i \ r)$

qualified primrec $\text{rpd}\kappa$ **where**

$\text{rpd}\kappa \ \kappa \ \text{test } i \ (\text{Skip } n) = (\text{case } n \ \text{of } 0 \Rightarrow \{\} \mid \text{Suc } m \Rightarrow \{\kappa \ (\text{Skip } m)\})$

$\text{rpd}\kappa \ \kappa \ \text{test } i \ (\text{Test } \varphi) = \{\}$

$\text{rpd}\kappa \ \kappa \ \text{test } i \ (\text{Plus } r \ s) = \text{rpd}\kappa \ \kappa \ \text{test } i \ r \cup \text{rpd}\kappa \ \kappa \ \text{test } i \ s$

$\text{rpd}\kappa \ \kappa \ \text{test } i \ (\text{Times } r \ s) = \text{rpd}\kappa \ (\lambda t. \kappa \ (\text{Times } r \ t)) \ \text{test } i \ s \cup (\text{if } \text{eps test } i \ s \ \text{then } \text{rpd}\kappa \ \kappa \ \text{test } i \ r \ \text{else } \{\})$

$\text{rpd}\kappa \ \kappa \ \text{test } i \ (\text{Star } r) = \text{rpd}\kappa \ (\lambda t. \kappa \ (\text{Times } (\text{Star } r) \ t)) \ \text{test } i \ r$

lemma $\text{lpd}\kappa_lpd$: $\text{lpd}\kappa \ \kappa \ \text{test } i \ r = \kappa \ ' \ \text{lpd test } i \ r$

<proof>

lemma $\text{rpd}\kappa_rpd$: $\text{rpd}\kappa \ \kappa \ \text{test } i \ r = \kappa \ ' \ \text{rpd test } i \ r$

<proof>

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

<proof>

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

<proof>

lemma eps_match : $\text{eps test } i \ r \longleftrightarrow \text{match test } r \ i \ i$

<proof>

lemma lpd_match : $i < j \implies \text{match test } r \ i \ j \longleftrightarrow (\bigsqcup s \in \text{lpd test } i \ r. \text{match test } s) \ (i + 1) \ j$

<proof>

lemma *rpd_match*: $i < j \implies \text{match test } r \text{ } i \text{ } j \iff (\bigsqcup s \in \text{rpd test } j \text{ } r. \text{match test } s) \text{ } i \text{ } (j - 1)$
<proof>

lemma *lpd_fv_regex*: $s \in \text{lpd test } i \text{ } r \implies \text{fv_regex } \text{fv } s \subseteq \text{fv_regex } \text{fv } r$
<proof>

lemma *rpd_fv_regex*: $s \in \text{rpd test } i \text{ } r \implies \text{fv_regex } \text{fv } s \subseteq \text{fv_regex } \text{fv } r$
<proof>

lemma *match_fv_cong*:
 $(\bigwedge i \ x. x \in \text{atms } r \implies \text{test } i \ x = \text{test}' \ i \ x) \implies \text{match test } r = \text{match test}' \ r$
<proof>

lemma *eps_fv_cong*:
 $(\bigwedge i \ x. x \in \text{atms } r \implies \text{test } i \ x = \text{test}' \ i \ x) \implies \text{eps test } i \ r = \text{eps test}' \ i \ r$
<proof>

datatype *modality* = *Past* | *Futu*
datatype *safety* = *Strict* | *Lax*

context
 fixes *fv* :: 'a \Rightarrow 'b set
 and *safe* :: *safety* \Rightarrow 'a \Rightarrow bool
begin

qualified fun *safe_regex* :: *modality* \Rightarrow *safety* \Rightarrow 'a *regex* \Rightarrow bool **where**
 safe_regex *m* _ (*Skip* *n*) = True
 | *safe_regex* *m* *g* (*Test* φ) = *safe* *g* φ
 | *safe_regex* *m* *g* (*Plus* *r* *s*) = ((*g* = *Lax* \vee *fv_regex* *fv* *r* = *fv_regex* *fv* *s*) \wedge *safe_regex* *m* *g* *r* \wedge *safe_regex* *m* *g* *s*)
 | *safe_regex* *Futu* *g* (*Times* *r* *s*) =
 ((*g* = *Lax* \vee *fv_regex* *fv* *r* \subseteq *fv_regex* *fv* *s*) \wedge *safe_regex* *Futu* *g* *s* \wedge *safe_regex* *Futu* *Lax* *r*)
 | *safe_regex* *Past* *g* (*Times* *r* *s*) =
 ((*g* = *Lax* \vee *fv_regex* *fv* *s* \subseteq *fv_regex* *fv* *r*) \wedge *safe_regex* *Past* *g* *r* \wedge *safe_regex* *Past* *Lax* *s*)
 | *safe_regex* *m* *g* (*Star* *r*) = ((*g* = *Lax* \vee *fv_regex* *fv* *r* = {}) \wedge *safe_regex* *m* *g* *r*)

lemmas *safe_regex_induct* = *safe_regex.induct*[*case_names* *Skip* *Test* *Plus* *TimesF* *TimesP* *Star*]

lemma *safe_cosafe*:
 $(\bigwedge x. x \in \text{atms } r \implies \text{safe } \text{Strict } x \implies \text{safe } \text{Lax } x) \implies \text{safe_regex } m \text{ } \text{Strict } r \implies \text{safe_regex } m \text{ } \text{Lax } r$
<proof>

lemma *safe_lpd_fv_regex_le*: $\text{safe_regex } \text{Futu } \text{Strict } r \implies s \in \text{lpd test } i \text{ } r \implies \text{fv_regex } \text{fv } r \subseteq \text{fv_regex } \text{fv } s$
<proof>

lemma *safe_lpd_fv_regex*: $\text{safe_regex } \text{Futu } \text{Strict } r \implies s \in \text{lpd test } i \text{ } r \implies \text{fv_regex } \text{fv } s = \text{fv_regex } \text{fv } r$
<proof>

lemma *cosafe_lpd*: $\text{safe_regex } \text{Futu } \text{Lax } r \implies s \in \text{lpd test } i \text{ } r \implies \text{safe_regex } \text{Futu } \text{Lax } s$
<proof>

lemma *safe_lpd*: $(\forall x \in \text{atms } r. \text{safe } \text{Strict } x \longrightarrow \text{safe } \text{Lax } x) \implies \text{safe_regex } \text{Futu } \text{Strict } r \implies s \in \text{lpd test } i \text{ } r \implies \text{safe_regex } \text{Futu } \text{Strict } s$
<proof>

lemma *safe_rpd_fv_regex_le*: *safe_regex Past Strict r* \implies *s* \in *rpd test i r* \implies *fv_regex fv r* \subseteq *fv_regex fv s*
 <proof>

lemma *safe_rpd_fv_regex*: *safe_regex Past Strict r* \implies *s* \in *rpd test i r* \implies *fv_regex fv s* = *fv_regex fv r*
 <proof>

lemma *cosafe_rpd*: *safe_regex Past Lax r* \implies *s* \in *rpd test i r* \implies *safe_regex Past Lax s*
 <proof>

lemma *safe_rpd*: $(\forall x \in \text{atms } r. \text{safe Strict } x \longrightarrow \text{safe Lax } x) \implies$
safe_regex Past Strict r \implies *s* \in *rpd test i r* \implies *safe_regex Past Strict s*
 <proof>

lemma *safe_regex_safe*: $(\bigwedge g r. \text{safe } g r \implies \text{safe Lax } r) \implies$
safe_regex m g r \implies *x* \in *atms r* \implies *safe Lax x*
 <proof>

lemma *safe_regex_map_regex*:
 $(\bigwedge g x. x \in \text{atms } r \implies \text{safe } g x \implies \text{safe } g (f x)) \implies (\bigwedge x. x \in \text{atms } r \implies \text{fv } (f x) = \text{fv } x) \implies$
safe_regex m g r \implies *safe_regex m g (map_regex f r)*
 <proof>

end

lemma *safe_regex_cong[fundef_cong]*:
 $(\bigwedge g x. x \in \text{atms } r \implies \text{safe } g x = \text{safe}' g x) \implies$
Regex.safe_regex fv safe m g r = *Regex.safe_regex fv safe' m g r*
 <proof>

lemma *safe_regex_mono*:
 $(\bigwedge g x. x \in \text{atms } r \implies \text{safe } g x \implies \text{safe}' g x) \implies$
Regex.safe_regex fv safe m g r \implies *Regex.safe_regex fv safe' m g r*
 <proof>

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

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

end

4 Metric first-order dynamic logic

derive *(eq) ceq enat*

instantiation *enat* :: *ccompare begin*

definition *ccompare_enat* :: *enat comparator option where*

ccompare_enat = *Some* $(\lambda x y. \text{if } x = y \text{ then } \text{order.Eq} \text{ else if } x < y \text{ then } \text{order.Lt} \text{ else } \text{order.Gt})$

instance <proof>

end

context begin

4.1 Formulas and satisfiability

qualified type_synonym *name* = *String.literal*

qualified type_synonym *event* = (*name* × *event_data list*)

qualified type_synonym *database* = (*name*, *event_data list set list*) *mapping*

qualified type_synonym *prefix* = (*name* × *event_data list*) *prefix*

qualified type_synonym *trace* = (*name* × *event_data list*) *trace*

qualified type_synonym *env* = *event_data list*

4.1.1 Syntax

qualified datatype *trm* = *is_Var: Var nat* | *is_Const: Const event_data*

| *Plus trm trm* | *Minus trm trm* | *UMinus trm* | *Mult trm trm* | *Div trm trm* | *Mod trm trm*

| *F2i trm* | *I2f trm*

qualified primrec *fvi_trm* :: *nat* ⇒ *trm* ⇒ *nat set* **where**

fvi_trm *b* (*Var x*) = (*if b* ≤ *x* then {*x* - *b*} else {})

| *fvi_trm* *b* (*Const* _) = {}

| *fvi_trm* *b* (*Plus x y*) = *fvi_trm* *b* *x* ∪ *fvi_trm* *b* *y*

| *fvi_trm* *b* (*Minus x y*) = *fvi_trm* *b* *x* ∪ *fvi_trm* *b* *y*

| *fvi_trm* *b* (*UMinus x*) = *fvi_trm* *b* *x*

| *fvi_trm* *b* (*Mult x y*) = *fvi_trm* *b* *x* ∪ *fvi_trm* *b* *y*

| *fvi_trm* *b* (*Div x y*) = *fvi_trm* *b* *x* ∪ *fvi_trm* *b* *y*

| *fvi_trm* *b* (*Mod x y*) = *fvi_trm* *b* *x* ∪ *fvi_trm* *b* *y*

| *fvi_trm* *b* (*F2i x*) = *fvi_trm* *b* *x*

| *fvi_trm* *b* (*I2f x*) = *fvi_trm* *b* *x*

abbreviation *fv_trm* ≡ *fvi_trm* 0

qualified primrec *eval_trm* :: *env* ⇒ *trm* ⇒ *event_data* **where**

eval_trm *v* (*Var x*) = *v* ! *x*

| *eval_trm* *v* (*Const x*) = *x*

| *eval_trm* *v* (*Plus x y*) = *eval_trm* *v* *x* + *eval_trm* *v* *y*

| *eval_trm* *v* (*Minus x y*) = *eval_trm* *v* *x* - *eval_trm* *v* *y*

| *eval_trm* *v* (*UMinus x*) = - *eval_trm* *v* *x*

| *eval_trm* *v* (*Mult x y*) = *eval_trm* *v* *x* * *eval_trm* *v* *y*

| *eval_trm* *v* (*Div x y*) = *eval_trm* *v* *x* div *eval_trm* *v* *y*

| *eval_trm* *v* (*Mod x y*) = *eval_trm* *v* *x* mod *eval_trm* *v* *y*

| *eval_trm* *v* (*F2i x*) = *EInt* (*integer_of_event_data* (*eval_trm* *v* *x*))

| *eval_trm* *v* (*I2f x*) = *EFloat* (*double_of_event_data* (*eval_trm* *v* *x*))

lemma *eval_trm_fv_cong*: ∀ *x* ∈ *fv_trm* *t*. *v* ! *x* = *v'* ! *x* ⇒ *eval_trm* *v* *t* = *eval_trm* *v'* *t*

⟨*proof*⟩ **datatype** *agg_type* = *Agg_Cnt* | *Agg_Min* | *Agg_Max* | *Agg_Sum* | *Agg_Avg* | *Agg_Med*

qualified type_synonym *agg_op* = *agg_type* × *event_data*

definition *flatten_multiset* :: (*event_data* × *enat*) *set* ⇒ *event_data list* **where**

flatten_multiset *M* = *concat* (*map* (λ(*x*, *c*). *replicate* (*the_enat* *c*) *x*) (*csorted_list_of_set* *M*))

fun *eval_agg_op* :: *agg_op* ⇒ (*event_data* × *enat*) *set* ⇒ *event_data* **where**

eval_agg_op (*Agg_Cnt*, *y0*) *M* = *EInt* (*integer_of_int* (*length* (*flatten_multiset* *M*)))

| *eval_agg_op* (*Agg_Min*, *y0*) *M* = (*case* *flatten_multiset* *M* of

[] ⇒ *y0*

| *x* # *xs* ⇒ *foldl* *min* *x* *xs*)

| *eval_agg_op* (*Agg_Max*, *y0*) *M* = (*case* *flatten_multiset* *M* of

[] ⇒ *y0*

| *x* # *xs* ⇒ *foldl* *max* *x* *xs*)

$| \text{eval_agg_op } (\text{Agg_Sum}, y0) M = \text{foldl plus } y0 (\text{flatten_multiset } M)$
 $| \text{eval_agg_op } (\text{Agg_Avg}, y0) M = \text{EFloat } (\text{let } xs = \text{flatten_multiset } M \text{ in case } xs \text{ of}$
 $\quad [] \Rightarrow 0$
 $\quad | _ \Rightarrow \text{double_of_event_data } (\text{foldl plus } (\text{EInt } 0) xs) / \text{double_of_int } (\text{length } xs))$
 $| \text{eval_agg_op } (\text{Agg_Med}, y0) M = \text{EFloat } (\text{let } xs = \text{flatten_multiset } M; u = \text{length } xs \text{ in}$
 $\quad \text{if } u = 0 \text{ then } 0 \text{ else}$
 $\quad \quad \text{let } u' = u \text{ div } 2 \text{ in}$
 $\quad \quad \text{if even } u \text{ then}$
 $\quad \quad \quad (\text{double_of_event_data } (xs ! (u'-1)) + \text{double_of_event_data } (xs ! u')) / \text{double_of_int } 2$
 $\quad \quad \text{else } \text{double_of_event_data } (xs ! u')$

qualified datatype (*discs_sels*) *formula* = *Pred name trm list*

$| \text{Let name formula formula}$
 $| \text{Eq trm trm} \mid \text{Less trm trm} \mid \text{LessEq trm trm}$
 $| \text{Neg formula} \mid \text{Or formula formula} \mid \text{And formula formula} \mid \text{Ands formula list} \mid \text{Exists formula}$
 $| \text{Agg nat agg_op nat trm formula}$
 $| \text{Prev } \mathcal{I} \text{ formula} \mid \text{Next } \mathcal{I} \text{ formula}$
 $| \text{Since formula } \mathcal{I} \text{ formula} \mid \text{Until formula } \mathcal{I} \text{ formula}$
 $| \text{MatchF } \mathcal{I} \text{ formula } \text{Regex.regex} \mid \text{MatchP } \mathcal{I} \text{ formula } \text{Regex.regex}$

qualified definition $FF = \text{Exists } (\text{Neg } (\text{Eq } (\text{Var } 0) (\text{Var } 0)))$

qualified definition $TT \equiv \text{Neg } FF$

qualified fun $fvi :: \text{nat} \Rightarrow \text{formula} \Rightarrow \text{nat set where}$

$fvi \ b \ (\text{Pred } r \ ts) = (\bigcup t \in \text{set } ts. \text{fvi_trm } b \ t)$
 $| fvi \ b \ (\text{Let } p \ \varphi \ \psi) = fvi \ b \ \psi$
 $| fvi \ b \ (\text{Eq } t1 \ t2) = \text{fvi_trm } b \ t1 \cup \text{fvi_trm } b \ t2$
 $| fvi \ b \ (\text{Less } t1 \ t2) = \text{fvi_trm } b \ t1 \cup \text{fvi_trm } b \ t2$
 $| fvi \ b \ (\text{LessEq } t1 \ t2) = \text{fvi_trm } b \ t1 \cup \text{fvi_trm } b \ t2$
 $| fvi \ b \ (\text{Neg } \varphi) = fvi \ b \ \varphi$
 $| fvi \ b \ (\text{Or } \varphi \ \psi) = fvi \ b \ \varphi \cup fvi \ b \ \psi$
 $| fvi \ b \ (\text{And } \varphi \ \psi) = fvi \ b \ \varphi \cup fvi \ b \ \psi$
 $| fvi \ b \ (\text{Ands } \varphi s) = (\text{let } xs = \text{map } (fvi \ b) \ \varphi s \text{ in } \bigcup x \in \text{set } xs. x)$
 $| fvi \ b \ (\text{Exists } \varphi) = fvi \ (\text{Suc } b) \ \varphi$
 $| fvi \ b \ (\text{Agg } y \ \omega \ b' \ f \ \varphi) = fvi \ (b + b') \ \varphi \cup \text{fvi_trm } (b + b') \ f \cup (\text{if } b \leq y \text{ then } \{y - b\} \text{ else } \{\})$
 $| fvi \ b \ (\text{Prev } I \ \varphi) = fvi \ b \ \varphi$
 $| fvi \ b \ (\text{Next } I \ \varphi) = fvi \ b \ \varphi$
 $| fvi \ b \ (\text{Since } \varphi \ I \ \psi) = fvi \ b \ \varphi \cup fvi \ b \ \psi$
 $| fvi \ b \ (\text{Until } \varphi \ I \ \psi) = fvi \ b \ \varphi \cup fvi \ b \ \psi$
 $| fvi \ b \ (\text{MatchF } I \ r) = \text{Regex.fv_regex } (fvi \ b) \ r$
 $| fvi \ b \ (\text{MatchP } I \ r) = \text{Regex.fv_regex } (fvi \ b) \ r$

abbreviation $fv \equiv fvi \ 0$

abbreviation $fv_regex \equiv \text{Regex.fv_regex } fv$

lemma $fv_abbrevs[simp]: fv \ TT = \{\} \quad fv \ FF = \{\}$
 $\langle \text{proof} \rangle$

lemma $fv_subset_Ands: \varphi \in \text{set } \varphi s \implies fv \ \varphi \subseteq fv \ (\text{Ands } \varphi s)$
 $\langle \text{proof} \rangle$

lemma $finite_fvi_trm[simp]: finite \ (\text{fvi_trm } b \ t)$
 $\langle \text{proof} \rangle$

lemma $finite_fvi[simp]: finite \ (fvi \ b \ \varphi)$
 $\langle \text{proof} \rangle$

lemma $fvi_trm_plus: x \in \text{fvi_trm } (b + c) \ t \longleftrightarrow x + c \in \text{fvi_trm } b \ t$

<proof>

lemma *fvi_trm_iff_fv_trm*: $x \in \text{fvi_trm } b \ t \longleftrightarrow x + b \in \text{fv_trm } t$
<proof>

lemma *fvi_plus*: $x \in \text{fvi } (b + c) \ \varphi \longleftrightarrow x + c \in \text{fvi } b \ \varphi$
<proof>

lemma *fvi_Suc*: $x \in \text{fvi } (\text{Suc } b) \ \varphi \longleftrightarrow \text{Suc } x \in \text{fvi } b \ \varphi$
<proof>

lemma *fvi_plus_bound*:
assumes $\forall i \in \text{fvi } (b + c) \ \varphi. \ i < n$
shows $\forall i \in \text{fvi } b \ \varphi. \ i < c + n$
<proof>

lemma *fvi_Suc_bound*:
assumes $\forall i \in \text{fvi } (\text{Suc } b) \ \varphi. \ i < n$
shows $\forall i \in \text{fvi } b \ \varphi. \ i < \text{Suc } n$
<proof>

lemma *fvi_iff_fv*: $x \in \text{fvi } b \ \varphi \longleftrightarrow x + b \in \text{fv } \varphi$
<proof> **definition** *nfv* :: *formula* \Rightarrow *nat* **where**
 $\text{nfv } \varphi = \text{Max } (\text{insert } 0 \ (\text{Suc } \text{'fv } \varphi))$

qualified abbreviation *nfv_regex* **where**
 $\text{nfv_regex} \equiv \text{Regex.nfv_regex } \text{fv}$

qualified definition *envs* :: *formula* \Rightarrow *env set* **where**
 $\text{envs } \varphi = \{v. \text{length } v = \text{nfv } \varphi\}$

lemma *nfv_simps[simp]*:
 $\text{nfv } (\text{Let } p \ \varphi \ \psi) = \text{nfv } \psi$
 $\text{nfv } (\text{Neg } \varphi) = \text{nfv } \varphi$
 $\text{nfv } (\text{Or } \varphi \ \psi) = \max (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{And } \varphi \ \psi) = \max (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{Prev } I \ \varphi) = \text{nfv } \varphi$
 $\text{nfv } (\text{Next } I \ \varphi) = \text{nfv } \varphi$
 $\text{nfv } (\text{Since } \varphi \ I \ \psi) = \max (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{Until } \varphi \ I \ \psi) = \max (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{MatchP } I \ r) = \text{Regex.nfv_regex } \text{fv } r$
 $\text{nfv } (\text{MatchF } I \ r) = \text{Regex.nfv_regex } \text{fv } r$
 $\text{nfv_regex } (\text{Regex.Skip } n) = 0$
 $\text{nfv_regex } (\text{Regex.Test } \varphi) = \text{Max } (\text{insert } 0 \ (\text{Suc } \text{'fv } \varphi))$
 $\text{nfv_regex } (\text{Regex.Plus } r \ s) = \max (\text{nfv_regex } r) (\text{nfv_regex } s)$
 $\text{nfv_regex } (\text{Regex.Times } r \ s) = \max (\text{nfv_regex } r) (\text{nfv_regex } s)$
 $\text{nfv_regex } (\text{Regex.Star } r) = \text{nfv_regex } r$
<proof>

lemma *nfv_Ands[simp]*: $\text{nfv } (\text{Ands } l) = \text{Max } (\text{insert } 0 \ (\text{nfv } \text{'set } l))$
<proof>

lemma *fvi_less_nfv*: $\forall i \in \text{fv } \varphi. \ i < \text{nfv } \varphi$
<proof>

lemma *fvi_less_nfv_regex*: $\forall i \in \text{fv_regex } \varphi. \ i < \text{nfv_regex } \varphi$
<proof>

4.1.2 Future reach

qualified fun *future_bounded* :: formula ⇒ bool **where**

future_bounded (Pred _ _) = True
 | *future_bounded* (Let p φ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
 | *future_bounded* (Eq _ _) = True
 | *future_bounded* (Less _ _) = True
 | *future_bounded* (LessEq _ _) = True
 | *future_bounded* (Neg φ) = *future_bounded* φ
 | *future_bounded* (Or φ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
 | *future_bounded* (And φ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
 | *future_bounded* (Ands l) = list_all *future_bounded* l
 | *future_bounded* (Exists φ) = *future_bounded* φ
 | *future_bounded* (Agg y ω b f φ) = *future_bounded* φ
 | *future_bounded* (Prev I φ) = *future_bounded* φ
 | *future_bounded* (Next I φ) = *future_bounded* φ
 | *future_bounded* (Since φ I ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
 | *future_bounded* (Until φ I ψ) = (*future_bounded* φ ∧ *future_bounded* ψ ∧ right I ≠ ∞)
 | *future_bounded* (MatchP I r) = Regex.pred_regex *future_bounded* r
 | *future_bounded* (MatchF I r) = (Regex.pred_regex *future_bounded* r ∧ right I ≠ ∞)

4.1.3 Semantics

definition *ecard* A = (if finite A then card A else ∞)

qualified fun *sat* :: trace ⇒ (name → nat ⇒ event_data list set) ⇒ env ⇒ nat ⇒ formula ⇒ bool **where**

sat σ V v i (Pred r ts) = (case V r of
 None ⇒ (r, map (eval_trm v) ts) ∈ Γ σ i
 | Some X ⇒ map (eval_trm v) ts ∈ X i)
 | *sat* σ V v i (Let p φ ψ) =
sat σ (V(p ↦ λi. {v. length v = nfv φ ∧ sat σ V v i φ})) v i ψ
 | *sat* σ V v i (Eq t1 t2) = (eval_trm v t1 = eval_trm v t2)
 | *sat* σ V v i (Less t1 t2) = (eval_trm v t1 < eval_trm v t2)
 | *sat* σ V v i (LessEq t1 t2) = (eval_trm v t1 ≤ eval_trm v t2)
 | *sat* σ V v i (Neg φ) = (¬ sat σ V v i φ)
 | *sat* σ V v i (Or φ ψ) = (sat σ V v i φ ∨ sat σ V v i ψ)
 | *sat* σ V v i (And φ ψ) = (sat σ V v i φ ∧ sat σ V v i ψ)
 | *sat* σ V v i (Ands l) = (∀ φ ∈ set l. sat σ V v i φ)
 | *sat* σ V v i (Exists φ) = (∃ z. sat σ V (z # v) i φ)
 | *sat* σ V v i (Agg y ω b f φ) =
 (let M = {(x, ecard Zs) | x Zs. Zs = {zs. length zs = b ∧ sat σ V (zs @ v) i φ ∧ eval_trm (zs @ v) f
 = x} ∧ Zs ≠ {}}
 in (M = {}) → fv φ ⊆ {0..<b} ∧ v ! y = eval_agg_op ω M)
 | *sat* σ V v i (Prev I φ) = (case i of 0 ⇒ False | Suc j ⇒ mem (τ σ i - τ σ j) I ∧ sat σ V v j φ)
 | *sat* σ V v i (Next I φ) = (mem (τ σ (Suc i) - τ σ i) I ∧ sat σ V v (Suc i) φ)
 | *sat* σ V v i (Since φ I ψ) = (∃ j ≤ i. mem (τ σ i - τ σ j) I ∧ sat σ V v j ψ ∧ (∀ k ∈ {j <.. i}. sat σ V
 v k φ))
 | *sat* σ V v i (Until φ I ψ) = (∃ j ≥ i. mem (τ σ j - τ σ i) I ∧ sat σ V v j ψ ∧ (∀ k ∈ {i <.. j}. sat σ V
 v k φ))
 | *sat* σ V v i (MatchP I r) = (∃ j ≤ i. mem (τ σ i - τ σ j) I ∧ Regex.match (sat σ V v) r j i)
 | *sat* σ V v i (MatchF I r) = (∃ j ≥ i. mem (τ σ j - τ σ i) I ∧ Regex.match (sat σ V v) r i j)

lemma *sat_abbrevs[simp]*:

sat σ V v i TT ¬ sat σ V v i FF
 ⟨proof⟩

lemma *sat_Ands*: *sat* σ V v i (Ands l) ↔ (∀ φ ∈ set l. sat σ V v i φ)

⟨proof⟩

lemma *sat_Until_rec*: $\text{sat } \sigma \ V \ v \ i \ (\text{Until } \varphi \ I \ \psi) \longleftrightarrow$
 $\text{mem } 0 \ I \wedge \text{sat } \sigma \ V \ v \ i \ \psi \vee$
 $(\Delta \sigma \ (i + 1) \leq \text{right } I \wedge \text{sat } \sigma \ V \ v \ i \ \varphi \wedge \text{sat } \sigma \ V \ v \ (i + 1) \ (\text{Until } \varphi \ (\text{subtract } (\Delta \sigma \ (i + 1)) \ I) \ \psi))$
 $(\text{is } ?L \longleftrightarrow ?R)$
 $\langle \text{proof} \rangle$

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

lemma *sat_MatchF_rec*: $\text{sat } \sigma \ V \ v \ i \ (\text{MatchF } I \ r) \longleftrightarrow \text{mem } 0 \ I \wedge \text{Regex.eps} \ (\text{sat } \sigma \ V \ v) \ i \ r \vee$
 $\Delta \sigma \ (i + 1) \leq \text{right } I \wedge (\exists s \in \text{Regex.lpd} \ (\text{sat } \sigma \ V \ v) \ i \ r. \text{sat } \sigma \ V \ v \ (i + 1) \ (\text{MatchF} \ (\text{subtract} \ (\Delta \sigma \ (i + 1)) \ I) \ s))$
 $(\text{is } ?L \longleftrightarrow ?R1 \vee ?R2)$
 $\langle \text{proof} \rangle$

lemma *sat_MatchP_rec*: $\text{sat } \sigma \ V \ v \ i \ (\text{MatchP } I \ r) \longleftrightarrow \text{mem } 0 \ I \wedge \text{Regex.eps} \ (\text{sat } \sigma \ V \ v) \ i \ r \vee$
 $i > 0 \wedge \Delta \sigma \ i \leq \text{right } I \wedge (\exists s \in \text{Regex.rpd} \ (\text{sat } \sigma \ V \ v) \ i \ r. \text{sat } \sigma \ V \ v \ (i - 1) \ (\text{MatchP} \ (\text{subtract} \ (\Delta \sigma \ i) \ I) \ s))$
 $(\text{is } ?L \longleftrightarrow ?R1 \vee ?R2)$
 $\langle \text{proof} \rangle$

lemma *sat_Since_0*: $\text{sat } \sigma \ V \ v \ 0 \ (\text{Since } \varphi \ I \ \psi) \longleftrightarrow \text{mem } 0 \ I \wedge \text{sat } \sigma \ V \ v \ 0 \ \psi$
 $\langle \text{proof} \rangle$

lemma *sat_MatchP_0*: $\text{sat } \sigma \ V \ v \ 0 \ (\text{MatchP } I \ r) \longleftrightarrow \text{mem } 0 \ I \wedge \text{Regex.eps} \ (\text{sat } \sigma \ V \ v) \ 0 \ r$
 $\langle \text{proof} \rangle$

lemma *sat_Since_point*: $\text{sat } \sigma \ V \ v \ i \ (\text{Since } \varphi \ I \ \psi) \implies$
 $(\bigwedge j. j \leq i \implies \text{mem} \ (\tau \sigma \ i - \tau \sigma \ j) \ I \implies \text{sat } \sigma \ V \ v \ i \ (\text{Since } \varphi \ (\text{point} \ (\tau \sigma \ i - \tau \sigma \ j)) \ \psi) \implies P)$
 $\implies P$
 $\langle \text{proof} \rangle$

lemma *sat_MatchP_point*: $\text{sat } \sigma \ V \ v \ i \ (\text{MatchP } I \ r) \implies$
 $(\bigwedge j. j \leq i \implies \text{mem} \ (\tau \sigma \ i - \tau \sigma \ j) \ I \implies \text{sat } \sigma \ V \ v \ i \ (\text{MatchP} \ (\text{point} \ (\tau \sigma \ i - \tau \sigma \ j)) \ r) \implies P)$
 $\implies P$
 $\langle \text{proof} \rangle$

lemma *sat_Since_pointD*: $\text{sat } \sigma \ V \ v \ i \ (\text{Since } \varphi \ (\text{point } t) \ \psi) \implies \text{mem } t \ I \implies \text{sat } \sigma \ V \ v \ i \ (\text{Since } \varphi \ I \ \psi)$
 $\langle \text{proof} \rangle$

lemma *sat_MatchP_pointD*: $\text{sat } \sigma \ V \ v \ i \ (\text{MatchP} \ (\text{point } t) \ r) \implies \text{mem } t \ I \implies \text{sat } \sigma \ V \ v \ i \ (\text{MatchP } I \ r)$
 $\langle \text{proof} \rangle$

lemma *sat_fv_cong*: $\forall x \in \text{fv } \varphi. v!x = v'!x \implies \text{sat } \sigma \ V \ v \ i \ \varphi = \text{sat } \sigma \ V \ v' \ i \ \varphi$
 $\langle \text{proof} \rangle$

lemma *match_fv_cong*:
 $\forall x \in \text{fv_regex } r. v!x = v'!x \implies \text{Regex.match} \ (\text{sat } \sigma \ V \ v) \ r = \text{Regex.match} \ (\text{sat } \sigma \ V \ v') \ r$
 $\langle \text{proof} \rangle$

lemma *eps_fv_cong*:
 $\forall x \in \text{fv_regex } r. v!x = v'!x \implies \text{Regex.eps} \ (\text{sat } \sigma \ V \ v) \ i \ r = \text{Regex.eps} \ (\text{sat } \sigma \ V \ v') \ i \ r$
 $\langle \text{proof} \rangle$

4.2 Past-only formulas

```

fun past_only :: formula  $\Rightarrow$  bool where
  past_only (Pred _ _) = True
| past_only (Eq _ _) = True
| past_only (Less _ _) = True
| past_only (LessEq _ _) = True
| past_only (Let _  $\alpha$   $\beta$ ) = (past_only  $\alpha$   $\wedge$  past_only  $\beta$ )
| past_only (Neg  $\psi$ ) = past_only  $\psi$ 
| past_only (Or  $\alpha$   $\beta$ ) = (past_only  $\alpha$   $\wedge$  past_only  $\beta$ )
| past_only (And  $\alpha$   $\beta$ ) = (past_only  $\alpha$   $\wedge$  past_only  $\beta$ )
| past_only (Ands l) = ( $\forall \alpha \in \text{set } l. \text{past\_only } \alpha$ )
| past_only (Exists  $\psi$ ) = past_only  $\psi$ 
| past_only (Agg _ _ _ _  $\psi$ ) = past_only  $\psi$ 
| past_only (Prev _  $\psi$ ) = past_only  $\psi$ 
| past_only (Next _ _) = False
| past_only (Since  $\alpha$  _  $\beta$ ) = (past_only  $\alpha$   $\wedge$  past_only  $\beta$ )
| past_only (Until  $\alpha$  _  $\beta$ ) = False
| past_only (MatchP _ r) = Regex.pred_regex past_only r
| past_only (MatchF _ _) = False

```

```

lemma past_only_sat:
  assumes prefix_of  $\pi$   $\sigma$  prefix_of  $\pi$   $\sigma'$ 
  shows  $i < \text{plen } \pi \implies \text{dom } V = \text{dom } V' \implies$ 
    ( $\bigwedge p i. p \in \text{dom } V \implies i < \text{plen } \pi \implies \text{the } (V p) i = \text{the } (V' p) i$ )  $\implies$ 
    past_only  $\varphi \implies \text{sat } \sigma V v i \varphi = \text{sat } \sigma' V' v i \varphi$ 
  <proof>

```

4.3 Safe formulas

```

fun remove_neg :: formula  $\Rightarrow$  formula where
  remove_neg (Neg  $\varphi$ ) =  $\varphi$ 
| remove_neg  $\varphi$  =  $\varphi$ 

```

```

lemma fvi_remove_neg[simp]: fvi b (remove_neg  $\varphi$ ) = fvi b  $\varphi$ 
  <proof>

```

```

lemma partition_cong[fundef_cong]:
   $xs = ys \implies (\bigwedge x. x \in \text{set } xs \implies f x = g x) \implies \text{partition } f xs = \text{partition } g ys$ 
  <proof>

```

```

lemma size_remove_neg[termination_simp]: size (remove_neg  $\varphi$ )  $\leq$  size  $\varphi$ 
  <proof>

```

```

fun is_constraint :: formula  $\Rightarrow$  bool where
  is_constraint (Eq t1 t2) = True
| is_constraint (Less t1 t2) = True
| is_constraint (LessEq t1 t2) = True
| is_constraint (Neg (Eq t1 t2)) = True
| is_constraint (Neg (Less t1 t2)) = True
| is_constraint (Neg (LessEq t1 t2)) = True
| is_constraint _ = False

```

```

definition safe_assignment :: nat set  $\Rightarrow$  formula  $\Rightarrow$  bool where
  safe_assignment X  $\varphi$  = (case  $\varphi$  of
    Eq (Var x) (Var y)  $\Rightarrow$  ( $x \notin X \longleftrightarrow y \in X$ )
  | Eq (Var x) t  $\Rightarrow$  ( $x \notin X \wedge \text{fv\_trm } t \subseteq X$ )
  | Eq t (Var x)  $\Rightarrow$  ( $x \notin X \wedge \text{fv\_trm } t \subseteq X$ )
  | _  $\Rightarrow$  False)

```

```

fun safe_formula :: formula  $\Rightarrow$  bool where
  safe_formula (Eq t1 t2) = (is_Const t1  $\wedge$  (is_Const t2  $\vee$  is_Var t2))  $\vee$  is_Var t1  $\wedge$  is_Const t2)
| safe_formula (Neg (Eq (Var x) (Var y))) = (x = y)
| safe_formula (Less t1 t2) = False
| safe_formula (LessEq t1 t2) = False
| safe_formula (Pred e ts) = ( $\forall t \in \text{set } ts. \text{is\_Var } t \vee \text{is\_Const } t$ )
| safe_formula (Let p  $\varphi$   $\psi$ ) = ( $\{0..<nfv \varphi\} \subseteq fv \varphi \wedge \text{safe\_formula } \varphi \wedge \text{safe\_formula } \psi$ )
| safe_formula (Neg  $\varphi$ ) = (fv  $\varphi$  =  $\{\}$ )  $\wedge$  safe_formula  $\varphi$ 
| safe_formula (Or  $\varphi$   $\psi$ ) = (fv  $\psi$  = fv  $\varphi$   $\wedge$  safe_formula  $\varphi$   $\wedge$  safe_formula  $\psi$ )
| safe_formula (And  $\varphi$   $\psi$ ) = (safe_formula  $\varphi$   $\wedge$ 
  (safe_assignment (fv  $\varphi$ )  $\psi$   $\vee$  safe_formula  $\psi$   $\vee$ 
  fv  $\psi \subseteq$  fv  $\varphi$   $\wedge$  (is_constraint  $\psi$   $\vee$  (case  $\psi$  of Neg  $\psi' \Rightarrow$  safe_formula  $\psi' | \_ \Rightarrow$  False))))
| safe_formula (Ands l) = (let (pos, neg) = partition safe_formula l in pos  $\neq$  []  $\wedge$ 
  list_all safe_formula (map remove_neg neg)  $\wedge$   $\bigcup$ (set (map fv neg))  $\subseteq$   $\bigcup$ (set (map fv pos)))
| safe_formula (Exists  $\varphi$ ) = (safe_formula  $\varphi$ )
| safe_formula (Agg y  $\omega$  b f  $\varphi$ ) = (safe_formula  $\varphi$   $\wedge$  y + b  $\notin$  fv  $\varphi$   $\wedge$   $\{0..<b\} \subseteq$  fv  $\varphi$   $\wedge$  fv_trm f  $\subseteq$  fv  $\varphi$ )
| safe_formula (Prev I  $\varphi$ ) = (safe_formula  $\varphi$ )
| safe_formula (Next I  $\varphi$ ) = (safe_formula  $\varphi$ )
| safe_formula (Since  $\varphi$  I  $\psi$ ) = (fv  $\varphi \subseteq$  fv  $\psi$   $\wedge$ 
  (safe_formula  $\varphi$   $\vee$  (case  $\varphi$  of Neg  $\varphi' \Rightarrow$  safe_formula  $\varphi' | \_ \Rightarrow$  False))  $\wedge$  safe_formula  $\psi$ )
| safe_formula (Until  $\varphi$  I  $\psi$ ) = (fv  $\varphi \subseteq$  fv  $\psi$   $\wedge$ 
  (safe_formula  $\varphi$   $\vee$  (case  $\varphi$  of Neg  $\varphi' \Rightarrow$  safe_formula  $\varphi' | \_ \Rightarrow$  False))  $\wedge$  safe_formula  $\psi$ )
| safe_formula (MatchP I r) = Regex.safe_regex fv ( $\lambda g \varphi. \text{safe\_formula } \varphi \vee$ 
  ( $g = \text{Lax} \wedge$  (case  $\varphi$  of Neg  $\varphi' \Rightarrow$  safe_formula  $\varphi' | \_ \Rightarrow$  False))) Past Strict r
| safe_formula (MatchF I r) = Regex.safe_regex fv ( $\lambda g \varphi. \text{safe\_formula } \varphi \vee$ 
  ( $g = \text{Lax} \wedge$  (case  $\varphi$  of Neg  $\varphi' \Rightarrow$  safe_formula  $\varphi' | \_ \Rightarrow$  False))) Futu Strict r

```

abbreviation safe_regex \equiv Regex.safe_regex fv ($\lambda g \varphi. \text{safe_formula } \varphi \vee$
($g = \text{Lax} \wedge$ (case φ of Neg $\varphi' \Rightarrow$ safe_formula $\varphi' | _ \Rightarrow$ False)))

lemma safe_regex_safe_formula:

```

safe_regex m g r  $\Longrightarrow$   $\varphi \in$  Regex.atms r  $\Longrightarrow$  safe_formula  $\varphi$   $\vee$ 
( $\exists \psi. \varphi = \text{Neg } \psi \wedge \text{safe\_formula } \psi$ )
<proof>

```

lemma safe_abbrevs[simp]: safe_formula TT safe_formula FF
<proof>

definition safe_neg :: formula \Rightarrow bool **where**

```

safe_neg  $\varphi \iff (\neg \text{safe\_formula } \varphi \longrightarrow \text{safe\_formula } (\text{remove\_neg } \varphi))$ 

```

definition atms :: formula Regex.regex \Rightarrow formula set **where**

```

atms r = ( $\bigcup \varphi \in$  Regex.atms r.
  if safe_formula  $\varphi$  then  $\{\varphi\}$  else case  $\varphi$  of Neg  $\varphi' \Rightarrow \{\varphi'\} | \_ \Rightarrow \{\}$ )

```

lemma atms_simps[simp]:

```

atms (Regex.Skip n) =  $\{\}$ 
atms (Regex.Test  $\varphi$ ) = (if safe_formula  $\varphi$  then  $\{\varphi\}$  else case  $\varphi$  of Neg  $\varphi' \Rightarrow \{\varphi'\} | \_ \Rightarrow \{\}$ )
atms (Regex.Plus r s) = atms r  $\cup$  atms s
atms (Regex.Times r s) = atms r  $\cup$  atms s
atms (Regex.Star r) = atms r
<proof>

```

lemma finite_atms[simp]: finite (atms r)

<proof>

lemma disjE_Not2: $P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (\neg P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$

<proof>

lemma *safe_formula_induct*[consumes 1, case_names Eq_Const Eq_Var1 Eq_Var2 neg_Var Pred Let And_assign And_safe And_constraint And_Not Ands Neg Or Exists Agg Prev Next Since Not_Since Until Not_Until MatchP MatchF]:

assumes *safe_formula* φ
and *Eq_Const*: $\bigwedge c d. P (Eq (Const c) (Const d))$
and *Eq_Var1*: $\bigwedge c x. P (Eq (Const c) (Var x))$
and *Eq_Var2*: $\bigwedge c x. P (Eq (Var x) (Const c))$
and *neg_Var*: $\bigwedge x. P (Neg (Eq (Var x) (Var x)))$
and *Pred*: $\bigwedge e ts. \forall t \in set ts. is_Var t \vee is_Const t \implies P (Pred e ts)$
and *Let*: $\bigwedge p \varphi \psi. \{0..<nfv \varphi\} \subseteq fv \varphi \implies safe_formula \varphi \implies safe_formula \psi \implies P \varphi \implies P \psi$
 $\implies P (Let p \varphi \psi)$
and *And_assign*: $\bigwedge \varphi \psi. safe_formula \varphi \implies safe_assignment (fv \varphi) \psi \implies P \varphi \implies P (And \varphi \psi)$
and *And_safe*: $\bigwedge \varphi \psi. safe_formula \varphi \implies \neg safe_assignment (fv \varphi) \psi \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (And \varphi \psi)$
and *And_constraint*: $\bigwedge \varphi \psi. safe_formula \varphi \implies \neg safe_assignment (fv \varphi) \psi \implies \neg safe_formula \psi \implies$
 $fv \psi \subseteq fv \varphi \implies is_constraint \psi \implies P \varphi \implies P (And \varphi \psi)$
and *And_Not*: $\bigwedge \varphi \psi. safe_formula \varphi \implies \neg safe_assignment (fv \varphi) (Neg \psi) \implies \neg safe_formula (Neg \psi) \implies$
 $fv (Neg \psi) \subseteq fv \varphi \implies \neg is_constraint (Neg \psi) \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (And \varphi (Neg \psi))$
and *Ands*: $\bigwedge l pos neg. (pos, neg) = partition safe_formula l \implies pos \neq [] \implies list_all safe_formula pos \implies list_all safe_formula (map remove_neg neg) \implies$
 $(\bigcup \varphi \in set neg. fv \varphi) \subseteq (\bigcup \varphi \in set pos. fv \varphi) \implies list_all P pos \implies list_all P (map remove_neg neg) \implies P (Ands l)$
and *Neg*: $\bigwedge \varphi. fv \varphi = \{\} \implies safe_formula \varphi \implies P \varphi \implies P (Neg \varphi)$
and *Or*: $\bigwedge \varphi \psi. fv \psi = fv \varphi \implies safe_formula \varphi \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (Or \varphi \psi)$
and *Exists*: $\bigwedge \varphi. safe_formula \varphi \implies P \varphi \implies P (Exists \varphi)$
and *Agg*: $\bigwedge y \omega b f \varphi. y + b \notin fv \varphi \implies \{0..<b\} \subseteq fv \varphi \implies fv_trm f \subseteq fv \varphi \implies safe_formula \varphi \implies P \varphi \implies P (Agg y \omega b f \varphi)$
and *Prev*: $\bigwedge I \varphi. safe_formula \varphi \implies P \varphi \implies P (Prev I \varphi)$
and *Next*: $\bigwedge I \varphi. safe_formula \varphi \implies P \varphi \implies P (Next I \varphi)$
and *Since*: $\bigwedge \varphi I \psi. fv \varphi \subseteq fv \psi \implies safe_formula \varphi \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (Since \varphi I \psi)$
and *Not_Since*: $\bigwedge \varphi I \psi. fv (Neg \varphi) \subseteq fv \psi \implies safe_formula \varphi \implies \neg safe_formula (Neg \varphi) \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (Since (Neg \varphi) I \psi)$
and *Until*: $\bigwedge \varphi I \psi. fv \varphi \subseteq fv \psi \implies safe_formula \varphi \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (Until \varphi I \psi)$
and *Not_Until*: $\bigwedge \varphi I \psi. fv (Neg \varphi) \subseteq fv \psi \implies safe_formula \varphi \implies \neg safe_formula (Neg \varphi) \implies safe_formula \psi \implies P \varphi \implies P \psi \implies P (Until (Neg \varphi) I \psi)$
and *MatchP*: $\bigwedge I r. safe_regex Past Strict r \implies \forall \varphi \in atms r. P \varphi \implies P (MatchP I r)$
and *MatchF*: $\bigwedge I r. safe_regex Futu Strict r \implies \forall \varphi \in atms r. P \varphi \implies P (MatchF I r)$
shows $P \varphi$
<proof>

lemma *safe_formula_NegD*:

$safe_formula (Formula.Neg \varphi) \implies fv \varphi = \{\} \vee (\exists x. \varphi = Formula.Eq (Formula.Var x) (Formula.Var x))$
<proof>

4.4 Slicing traces

qualified fun *matches* ::

$env \Rightarrow formula \Rightarrow name \times event_data list \Rightarrow bool$ **where**
 $matches v (Pred r ts) e = (fst e = r \wedge map (eval_trm v) ts = snd e)$

$| \text{matches } v \text{ (Let } p \ \varphi \ \psi) \ e =$
 $(\exists v'. \text{matches } v' \ \varphi \ e \wedge \text{matches } v \ \psi \ (p, v')) \vee$
 $\text{fst } e \neq p \wedge \text{matches } v \ \psi \ e)$
 $| \text{matches } v \text{ (Eq } _ _) \ e = \text{False}$
 $| \text{matches } v \text{ (Less } _ _) \ e = \text{False}$
 $| \text{matches } v \text{ (LessEq } _ _) \ e = \text{False}$
 $| \text{matches } v \text{ (Neg } \varphi) \ e = \text{matches } v \ \varphi \ e$
 $| \text{matches } v \text{ (Or } \varphi \ \psi) \ e = (\text{matches } v \ \varphi \ e \vee \text{matches } v \ \psi \ e)$
 $| \text{matches } v \text{ (And } \varphi \ \psi) \ e = (\text{matches } v \ \varphi \ e \wedge \text{matches } v \ \psi \ e)$
 $| \text{matches } v \text{ (Ands } l) \ e = (\exists \varphi \in \text{set } l. \text{matches } v \ \varphi \ e)$
 $| \text{matches } v \text{ (Exists } \varphi) \ e = (\exists z. \text{matches } (z \# v) \ \varphi \ e)$
 $| \text{matches } v \text{ (Agg } y \ \omega \ b \ f \ \varphi) \ e = (\exists zs. \text{length } zs = b \wedge \text{matches } (zs \ @ \ v) \ \varphi \ e)$
 $| \text{matches } v \text{ (Prev } I \ \varphi) \ e = \text{matches } v \ \varphi \ e$
 $| \text{matches } v \text{ (Next } I \ \varphi) \ e = \text{matches } v \ \varphi \ e$
 $| \text{matches } v \text{ (Since } \varphi \ I \ \psi) \ e = (\text{matches } v \ \varphi \ e \vee \text{matches } v \ \psi \ e)$
 $| \text{matches } v \text{ (Until } \varphi \ I \ \psi) \ e = (\text{matches } v \ \varphi \ e \vee \text{matches } v \ \psi \ e)$
 $| \text{matches } v \text{ (MatchP } I \ r) \ e = (\exists \varphi \in \text{Regex.atms } r. \text{matches } v \ \varphi \ e)$
 $| \text{matches } v \text{ (MatchF } I \ r) \ e = (\exists \varphi \in \text{Regex.atms } r. \text{matches } v \ \varphi \ e)$

lemma *matches_cong*:

$\forall x \in \text{fv } \varphi. v!x = v!x \implies \text{matches } v \ \varphi \ e = \text{matches } v' \ \varphi \ e$
<proof>

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

lemma *sat_slice_strong*:

assumes $v \in S \text{ dom } V = \text{dom } V'$

$\bigwedge p \ v \ i. p \in \text{dom } V \implies (p, v) \in \text{relevant_events } \varphi \ S \implies v \in \text{the } (V \ p) \ i \longleftrightarrow v \in \text{the } (V' \ p) \ i$

shows $\text{relevant_events } \varphi \ S - \{e. \text{fst } e \in \text{dom } V\} \subseteq E \implies$

$\text{sat } \sigma \ V \ v \ i \ \varphi \longleftrightarrow \text{sat } (\text{map_}\Gamma \ (\lambda D. D \cap E) \ \sigma) \ V' \ v \ i \ \varphi$

<proof>

4.5 Translation to n-ary conjunction

fun *get_and_list* :: *formula* \Rightarrow *formula list* **where**

get_and_list (Ands l) = l

$| \text{get_and_list } \varphi = [\varphi]$

lemma *fv_get_and*: $(\bigcup x \in (\text{set } (\text{get_and_list } \varphi)). \text{fv } b \ x) = \text{fv } b \ \varphi$

<proof>

lemma *safe_get_and*: $\text{safe_formula } \varphi \implies \text{list_all } \text{safe_neg } (\text{get_and_list } \varphi)$

<proof>

lemma *sat_get_and*: $\text{sat } \sigma \ V \ v \ i \ \varphi \longleftrightarrow \text{list_all } (\text{sat } \sigma \ V \ v \ i) (\text{get_and_list } \varphi)$

<proof>

fun *convert_multiway* :: *formula* \Rightarrow *formula* **where**

convert_multiway (Neg φ) = Neg (*convert_multiway* φ)

$| \text{convert_multiway } (\text{Or } \varphi \ \psi) = \text{Or } (\text{convert_multiway } \varphi) (\text{convert_multiway } \psi)$

$| \text{convert_multiway } (\text{And } \varphi \ \psi) = (\text{if } \text{safe_assignment } (\text{fv } \varphi) \ \psi \ \text{then}$

$\text{And } (\text{convert_multiway } \varphi) \ \psi$

$\text{else if } \text{safe_formula } \psi \ \text{then}$

$\text{Ands } (\text{get_and_list } (\text{convert_multiway } \varphi) \ @ \ \text{get_and_list } (\text{convert_multiway } \psi))$

$\text{else if } \text{is_constraint } \psi \ \text{then}$

$\text{And } (\text{convert_multiway } \varphi) \ \psi$

$\text{else Ands } (\text{convert_multiway } \psi \ \# \ \text{get_and_list } (\text{convert_multiway } \varphi))$

$| \text{convert_multiway } (\text{Exists } \varphi) = \text{Exists } (\text{convert_multiway } \varphi)$

$| \text{convert_multiway } (\text{Agg } y \ \omega \ b \ f \ \varphi) = \text{Agg } y \ \omega \ b \ f \ (\text{convert_multiway } \varphi)$
 $| \text{convert_multiway } (\text{Prev } I \ \varphi) = \text{Prev } I \ (\text{convert_multiway } \varphi)$
 $| \text{convert_multiway } (\text{Next } I \ \varphi) = \text{Next } I \ (\text{convert_multiway } \varphi)$
 $| \text{convert_multiway } (\text{Since } \varphi \ I \ \psi) = \text{Since } (\text{convert_multiway } \varphi) \ I \ (\text{convert_multiway } \psi)$
 $| \text{convert_multiway } (\text{Until } \varphi \ I \ \psi) = \text{Until } (\text{convert_multiway } \varphi) \ I \ (\text{convert_multiway } \psi)$
 $| \text{convert_multiway } (\text{MatchP } I \ r) = \text{MatchP } I \ (\text{Regex.map_regex } \text{convert_multiway } r)$
 $| \text{convert_multiway } (\text{MatchF } I \ r) = \text{MatchF } I \ (\text{Regex.map_regex } \text{convert_multiway } r)$
 $| \text{convert_multiway } \varphi = \varphi$

abbreviation $\text{convert_multiway_regex} \equiv \text{Regex.map_regex } \text{convert_multiway}$

lemma fv_safe_get_and :

$\text{safe_formula } \varphi \implies \text{fv } \varphi \subseteq (\bigcup x \in (\text{set } (\text{filter } \text{safe_formula } (\text{get_and_list } \varphi))). \text{fv } x)$
 $\langle \text{proof} \rangle$

lemma ex_safe_get_and :

$\text{safe_formula } \varphi \implies \text{list_ex } \text{safe_formula } (\text{get_and_list } \varphi)$
 $\langle \text{proof} \rangle$

lemma case_NegE : $(\text{case } \varphi \text{ of } \text{Neg } \varphi' \Rightarrow P \ \varphi' \mid _ \Rightarrow \text{False}) \implies (\bigwedge \varphi'. \varphi = \text{Neg } \varphi' \implies P \ \varphi' \implies Q) \implies Q$

$\langle \text{proof} \rangle$

lemma $\text{convert_multiway_remove_neg}$: $\text{safe_formula } (\text{remove_neg } \varphi) \implies \text{convert_multiway } (\text{remove_neg } \varphi) = \text{remove_neg } (\text{convert_multiway } \varphi)$

$\langle \text{proof} \rangle$

lemma $\text{fv_convert_multiway}$: $\text{safe_formula } \varphi \implies \text{fvi } b \ (\text{convert_multiway } \varphi) = \text{fvi } b \ \varphi$

$\langle \text{proof} \rangle$

lemma get_and_nonempty :

assumes $\text{safe_formula } \varphi$
shows $\text{get_and_list } \varphi \neq []$
 $\langle \text{proof} \rangle$

lemma $\text{future_bounded_get_and}$:

$\text{list_all } \text{future_bounded } (\text{get_and_list } \varphi) = \text{future_bounded } \varphi$
 $\langle \text{proof} \rangle$

lemma $\text{safe_convert_multiway}$: $\text{safe_formula } \varphi \implies \text{safe_formula } (\text{convert_multiway } \varphi)$

$\langle \text{proof} \rangle$

lemma $\text{future_bounded_convert_multiway}$: $\text{safe_formula } \varphi \implies \text{future_bounded } (\text{convert_multiway } \varphi) = \text{future_bounded } \varphi$

$\langle \text{proof} \rangle$

lemma $\text{sat_convert_multiway}$: $\text{safe_formula } \varphi \implies \text{sat } \sigma \ V \ v \ i \ (\text{convert_multiway } \varphi) \longleftrightarrow \text{sat } \sigma \ V \ v \ i \ \varphi$

$\langle \text{proof} \rangle$

end

interpretation Formula_slicer : $\text{abstract_slicer } \text{relevant_events } \varphi$ **for** φ $\langle \text{proof} \rangle$

lemma sat_slice_iff :

assumes $v \in S$
shows $\text{Formula.sat } \sigma \ V \ v \ i \ \varphi \longleftrightarrow \text{Formula.sat } (\text{Formula_slicer.slice } \varphi \ S \ \sigma) \ V \ v \ i \ \varphi$
 $\langle \text{proof} \rangle$

lemma *Neg_splits*:

P (case φ of formula.Neg $\psi \Rightarrow f \psi \mid \varphi \Rightarrow g \varphi$) =
 $((\forall \psi. \varphi = \text{formula.Neg } \psi \longrightarrow P (f \psi)) \wedge ((\neg \text{Formula.is_Neg } \varphi) \longrightarrow P (g \varphi)))$
 P (case φ of formula.Neg $\psi \Rightarrow f \psi \mid _ \Rightarrow g \varphi$) =
 $(\neg ((\exists \psi. \varphi = \text{formula.Neg } \psi \wedge \neg P (f \psi)) \vee ((\neg \text{Formula.is_Neg } \varphi) \wedge \neg P (g \varphi))))$
 <proof>

5 Optimized relational join

5.1 Binary join

definition *join_mask* :: $\text{nat} \Rightarrow \text{nat set} \Rightarrow \text{bool list}$ **where**
join_mask n $X = \text{map } (\lambda i. i \in X) [0..<n]$

fun *proj_tuple* :: $\text{bool list} \Rightarrow 'a \text{ tuple} \Rightarrow 'a \text{ tuple}$ **where**
proj_tuple [] [] = []
 | *proj_tuple* (True # bs) (a # as) = a # *proj_tuple* bs as
 | *proj_tuple* (False # bs) (a # as) = None # *proj_tuple* bs as
 | *proj_tuple* (b # bs) [] = []
 | *proj_tuple* [] (a # as) = []

lemma *proj_tuple_replicate*: $(\bigwedge i. i \in \text{set } bs \Longrightarrow \neg i) \Longrightarrow \text{length } bs = \text{length } as \Longrightarrow$
proj_tuple bs as = replicate (length bs) None
 <proof>

lemma *proj_tuple_join_mask_empty*: $\text{length } as = n \Longrightarrow$
proj_tuple (*join_mask* n {}) as = replicate n None
 <proof>

lemma *proj_tuple_alt*: *proj_tuple* bs as = map2 ($\lambda b a. \text{if } b \text{ then } a \text{ else None}$) bs as
 <proof>

lemma *map2_map*: map2 f (map g [0..<length as]) as = map ($\lambda i. f (g i) (as ! i)$) [0..<length as]
 <proof>

lemma *proj_tuple_join_mask_restrict*: $\text{length } as = n \Longrightarrow$
proj_tuple (*join_mask* n X) as = restrict X as
 <proof>

lemma *wf_tuple_proj_idle*:
assumes *wf*: *wf_tuple* n X as
shows *proj_tuple* (*join_mask* n X) as = as
 <proof>

lemma *wf_tuple_change_base*:
assumes *wf*: *wf_tuple* n X as
and *mask*: *join_mask* n $X = \text{join_mask } n$ Y
shows *wf_tuple* n Y as
 <proof>

definition *proj_tuple_in_join* :: $\text{bool} \Rightarrow \text{bool list} \Rightarrow 'a \text{ tuple} \Rightarrow 'a \text{ table} \Rightarrow \text{bool}$ **where**
proj_tuple_in_join pos bs as $t = (\text{if } pos \text{ then } \text{proj_tuple } bs \text{ as} \in t \text{ else } \text{proj_tuple } bs \text{ as} \notin t)$

abbreviation *join_cond* pos $t \equiv (\lambda as. \text{if } pos \text{ then } as \in t \text{ else } as \notin t)$

abbreviation *join_filter_cond* pos $t \equiv (\lambda as _. \text{join_cond } pos \text{ } t \text{ } as)$

lemma *proj_tuple_in_join_mask_idle*:

assumes *wf*: *wf_tuple* *n* *X* *as*

shows *proj_tuple_in_join* *pos* (*join_mask* *n* *X*) *as* *t* \longleftrightarrow *join_cond* *pos* *t* *as*

<proof>

lemma *join_sub*:

assumes $L \subseteq R$ *table* *n* *L* *t1* *table* *n* *R* *t2*

shows *join* *t2* *pos* *t1* = {*as* \in *t2*. *proj_tuple_in_join* *pos* (*join_mask* *n* *L*) *as* *t1*}

<proof>

lemma *join_sub'*:

assumes $R \subseteq L$ *table* *n* *L* *t1* *table* *n* *R* *t2*

shows *join* *t2* *True* *t1* = {*as* \in *t1*. *proj_tuple_in_join* *True* (*join_mask* *n* *R*) *as* *t2*}

<proof>

lemma *join_eq*:

assumes *tab*: *table* *n* *R* *t1* *table* *n* *R* *t2*

shows *join* *t2* *pos* *t1* = (if *pos* then $t2 \cap t1$ else $t2 - t1$)

<proof>

lemma *join_no_cols*:

assumes *tab*: *table* *n* {} *t1* *table* *n* *R* *t2*

shows *join* *t2* *pos* *t1* = (if (*pos* \longleftrightarrow *replicate* *n* *None* \in *t1*) then *t2* else {})

<proof>

lemma *join_empty_left*: *join* {} *pos* *t* = {}

<proof>

lemma *join_empty_right*: *join* *t* *pos* {} = (if *pos* then {} else *t*)

<proof>

fun *bin_join* :: *nat* \Rightarrow *nat* *set* \Rightarrow 'a *table* \Rightarrow *bool* \Rightarrow *nat* *set* \Rightarrow 'a *table* \Rightarrow 'a *table* **where**

bin_join *n* *A* *t* *pos* *A'* *t'* =

(if *t* = {} then {}

else if *t'* = {} then (if *pos* then {} else *t*)

else if *A'* = {} then (if (*pos* \longleftrightarrow *replicate* *n* *None* \in *t'*) then *t* else {})

else if *A'* = *A* then (if *pos* then $t \cap t'$ else $t - t'$)

else if $A' \subseteq A$ then {*as* \in *t*. *proj_tuple_in_join* *pos* (*join_mask* *n* *A'*) *as* *t'*}

else if $A \subseteq A' \wedge$ *pos* then {*as* \in *t'*. *proj_tuple_in_join* *pos* (*join_mask* *n* *A*) *as* *t*}

else *join* *t* *pos* *t'*)

lemma *bin_join_table*:

assumes *tab*: *table* *n* *A* *t* *table* *n* *A'* *t'*

shows *bin_join* *n* *A* *t* *pos* *A'* *t'* = *join* *t* *pos* *t'*

<proof>

5.2 Multi-way join

fun *mmulti_join'* :: (*nat* *set* *list* \Rightarrow *nat* *set* *list* \Rightarrow 'a *table* *list* \Rightarrow 'a *table*) **where**

mmulti_join' *A_pos* *A_neg* *L* = (

let *Q* = *set* (*zip* *A_pos* *L*) in

let *Q_neg* = *set* (*zip* *A_neg* (*drop* (*length* *A_pos*) *L*)) in

New_max_getIJ_wrapperGenericJoin *Q* *Q_neg*)

lemma *mmulti_join'_correct*:

assumes $A_pos \neq []$

and *list_all2* (λA *X*. *table* *n* *A* *X* \wedge *wf_set* *n* *A*) (*A_pos* @ *A_neg*) *L*

shows $z \in$ *mmulti_join'* *A_pos* *A_neg* *L* \longleftrightarrow *wf_tuple* *n* ($\bigcup_{A \in \text{set } A_pos. A}$) *z* \wedge

$list_all2 (\lambda A X. restrict A z \in X) A_pos (take (length A_pos) L) \wedge$
 $list_all2 (\lambda A X. restrict A z \notin X) A_neg (drop (length A_pos) L)$
 {proof}

lemmas $restrict_nested = New_max.restrict_nested$

lemma $list_all2_opt_True$:

assumes $list_all2 (\lambda A X. table n A X \wedge wf_set n A) ((A_zs @ A_x \# A_xs @ A_y \# A_ys) @ A_neg)$
 $((zs @ x \# xs @ y \# ys) @ L_neg)$
 $length A_xs = length xs \ length A_ys = length ys \ length A_zs = length zs$
shows $list_all2 (\lambda A X. table n A X \wedge wf_set n A)$
 $((A_zs @ (A_x \cup A_y) \# A_xs @ A_ys) @ A_neg) ((zs @ join x True y \# xs @ ys) @ L_neg)$
 {proof}

lemma $mmulti_join'_opt_True$:

assumes $list_all2 (\lambda A X. table n A X \wedge wf_set n A) ((A_zs @ A_x \# A_xs @ A_y \# A_ys) @ A_neg)$
 $((zs @ x \# xs @ y \# ys) @ L_neg)$
 $length A_xs = length xs \ length A_ys = length ys \ length A_zs = length zs$
shows $mmulti_join' (A_zs @ A_x \# A_xs @ A_y \# A_ys) A_neg ((zs @ x \# xs @ y \# ys) @ L_neg) =$
 $mmulti_join' (A_zs @ (A_x \cup A_y) \# A_xs @ A_ys) A_neg$
 $((zs @ join x True y \# xs @ ys) @ L_neg)$
 {proof}

lemma $list_all2_opt_False$:

assumes $list_all2 (\lambda A X. table n A X \wedge wf_set n A)$
 $((A_zs @ A_x \# A_xs) @ (A_ws @ A_y \# A_ys)) ((zs @ x \# xs) @ (ws @ y \# ys))$
 $length A_ws = length ws \ length A_xs = length xs$
 $length A_ys = length ys \ length A_zs = length zs$
 $A_y \subseteq A_x$
shows $list_all2 (\lambda A X. table n A X \wedge wf_set n A)$
 $((A_zs @ A_x \# A_xs) @ (A_ws @ A_ys)) ((zs @ join x False y \# xs) @ (ws @ ys))$
 {proof}

lemma $mmulti_join'_opt_False$:

assumes $list_all2 (\lambda A X. table n A X \wedge wf_set n A)$
 $((A_zs @ A_x \# A_xs) @ (A_ws @ A_y \# A_ys)) ((zs @ x \# xs) @ (ws @ y \# ys))$
 $length A_ws = length ws \ length A_xs = length xs$
 $length A_ys = length ys \ length A_zs = length zs$
 $A_y \subseteq A_x$
shows $mmulti_join' (A_zs @ A_x \# A_xs) (A_ws @ A_y \# A_ys) ((zs @ x \# xs) @ (ws @ y \# ys)) =$
 $mmulti_join' (A_zs @ A_x \# A_xs) (A_ws @ A_ys) ((zs @ join x False y \# xs) @ (ws @ ys))$
 {proof}

fun $find_sub_in :: 'a set \Rightarrow 'a set list \Rightarrow bool \Rightarrow$

$('a set list \times 'a set \times 'a set list) option$ **where**
 $find_sub_in X [] b = None$
 $| find_sub_in X (x \# xs) b = (if (x \subseteq X \vee (b \wedge X \subseteq x)) then Some ([], x, xs)$
 $else (case find_sub_in X xs b of None \Rightarrow None | Some (ys, z, zs) \Rightarrow Some (x \# ys, z, zs)))$

lemma $find_sub_in_sound: find_sub_in X xs b = Some (ys, z, zs) \Longrightarrow$

$xs = ys @ z \# zs \wedge (z \subseteq X \vee (b \wedge X \subseteq z))$
 {proof}

fun $find_sub_True :: 'a set list \Rightarrow$

$(\text{'a set list} \times \text{'a set} \times \text{'a set list} \times \text{'a set} \times \text{'a set list})$ option **where**
 $\text{find_sub_True } [] = \text{None}$
 $|\ \text{find_sub_True } (x \# xs) = (\text{case find_sub_in } x \text{ xs True of None } \Rightarrow$
 $\quad (\text{case find_sub_True } xs \text{ of None } \Rightarrow \text{None}$
 $\quad |\ \text{Some } (ys, w, ws, z, zs) \Rightarrow \text{Some } (x \# ys, w, ws, z, zs))$
 $|\ \text{Some } (ys, z, zs) \Rightarrow \text{Some } ([], x, ys, z, zs))$

lemma find_sub_True_sound: $\text{find_sub_True } xs = \text{Some } (ys, w, ws, z, zs) \Rightarrow$
 $xs = ys @ w \# ws @ z \# zs \wedge (z \subseteq w \vee w \subseteq z)$
 <proof>

fun find_sub_False :: 'a set list \Rightarrow 'a set list \Rightarrow
 $((\text{'a set list} \times \text{'a set} \times \text{'a set list}) \times (\text{'a set list} \times \text{'a set} \times \text{'a set list}))$ option **where**
 $\text{find_sub_False } [] \text{ ns} = \text{None}$
 $|\ \text{find_sub_False } (x \# xs) \text{ ns} = (\text{case find_sub_in } x \text{ ns False of None } \Rightarrow$
 $\quad (\text{case find_sub_False } xs \text{ ns of None } \Rightarrow \text{None}$
 $\quad |\ \text{Some } ((rs, w, ws), (ys, z, zs)) \Rightarrow \text{Some } ((x \# rs, w, ws), (ys, z, zs)))$
 $|\ \text{Some } (ys, z, zs) \Rightarrow \text{Some } ([], x, xs), (ys, z, zs)))$

lemma find_sub_False_sound: $\text{find_sub_False } xs \text{ ns} = \text{Some } ((rs, w, ws), (ys, z, zs)) \Rightarrow$
 $xs = rs @ w \# ws \wedge ns = ys @ z \# zs \wedge (z \subseteq w)$
 <proof>

fun proj_list_3 :: 'a list \Rightarrow ('b list \times 'b \times 'b list) \Rightarrow ('a list \times 'a \times 'a list) **where**
 $\text{proj_list_3 } xs \text{ (ys, z, zs)} = (\text{take } (\text{length } ys) \text{ xs}, xs ! (\text{length } ys),$
 $\quad \text{take } (\text{length } zs) \text{ (drop } (\text{length } ys + 1) \text{ xs)})$

lemma proj_list_3_same:
assumes $\text{proj_list_3 } xs \text{ (ys, z, zs)} = (ys', z', zs')$
 $\text{length } xs = \text{length } ys + 1 + \text{length } zs$
shows $xs = ys' @ z' \# zs'$
 <proof>

lemma proj_list_3_length:
assumes $\text{proj_list_3 } xs \text{ (ys, z, zs)} = (ys', z', zs')$
 $\text{length } xs = \text{length } ys + 1 + \text{length } zs$
shows $\text{length } ys = \text{length } ys' \text{ length } zs = \text{length } zs'$
 <proof>

fun proj_list_5 :: 'a list \Rightarrow
 $(\text{'b list} \times \text{'b} \times \text{'b list} \times \text{'b} \times \text{'b list}) \Rightarrow$
 $(\text{'a list} \times \text{'a} \times \text{'a list} \times \text{'a} \times \text{'a list})$ **where**
 $\text{proj_list_5 } xs \text{ (ys, w, ws, z, zs)} = (\text{take } (\text{length } ys) \text{ xs}, xs ! (\text{length } ys),$
 $\quad \text{take } (\text{length } ws) \text{ (drop } (\text{length } ys + 1) \text{ xs)}, xs ! (\text{length } ys + 1 + \text{length } ws),$
 $\quad \text{drop } (\text{length } ys + 1 + \text{length } ws + 1) \text{ xs})$

lemma proj_list_5_same:
assumes $\text{proj_list_5 } xs \text{ (ys, w, ws, z, zs)} = (ys', w', ws', z', zs')$
 $\text{length } xs = \text{length } ys + 1 + \text{length } ws + 1 + \text{length } zs$
shows $xs = ys' @ w' \# ws' @ z' \# zs'$
 <proof>

lemma proj_list_5_length:
assumes $\text{proj_list_5 } xs \text{ (ys, w, ws, z, zs)} = (ys', w', ws', z', zs')$
 $\text{length } xs = \text{length } ys + 1 + \text{length } ws + 1 + \text{length } zs$
shows $\text{length } ys = \text{length } ys' \text{ length } ws = \text{length } ws'$
 $\text{length } zs = \text{length } zs'$
 <proof>

fun *dominate_True* :: *nat set list* \Rightarrow *'a table list* \Rightarrow
 ((*nat set list* \times *nat set* \times *nat set list* \times *nat set* \times *nat set list*) \times
 (*'a table list* \times *'a table* \times *'a table list* \times *'a table* \times *'a table list*)) **option where**
dominate_True *A_pos* *L_pos* = (case *find_sub_True* *A_pos* of *None* \Rightarrow *None*
 | *Some split* \Rightarrow *Some (split, proj_list_5 L_pos split)*)

lemma *find_sub_True_proj_list_5_same*:
assumes *find_sub_True* *xs* = *Some (ys, w, ws, z, zs)* *length xs* = *length xs'*
proj_list_5 xs' (ys, w, ws, z, zs) = (*ys', w', ws', z', zs'*)
shows *xs' = ys' @ w' # ws' @ z' # zs'*
 <proof>

lemma *find_sub_True_proj_list_5_length*:
assumes *find_sub_True* *xs* = *Some (ys, w, ws, z, zs)* *length xs* = *length xs'*
proj_list_5 xs' (ys, w, ws, z, zs) = (*ys', w', ws', z', zs'*)
shows *length ys* = *length ys'* *length ws* = *length ws'*
length zs = *length zs'*
 <proof>

lemma *dominate_True_sound*:
assumes *dominate_True* *A_pos* *L_pos* = *Some ((A_zs, A_x, A_xs, A_y, A_ys), (zs, x, xs, y, ys))*
length A_pos = *length L_pos*
shows *A_pos* = *A_zs @ A_x # A_xs @ A_y # A_ys* *L_pos* = *zs @ x # xs @ y # ys*
length A_xs = *length xs* *length A_ys* = *length ys* *length A_zs* = *length zs*
 <proof>

fun *dominate_False* :: *nat set list* \Rightarrow *'a table list* \Rightarrow *nat set list* \Rightarrow *'a table list* \Rightarrow
 (((*nat set list* \times *nat set* \times *nat set list*) \times *nat set list* \times *nat set* \times *nat set list*) \times
 (*'a table list* \times *'a table* \times *'a table list*) \times
 (*'a table list* \times *'a table* \times *'a table list*)) **option where**
dominate_False *A_pos* *L_pos* *A_neg* *L_neg* = (case *find_sub_False* *A_pos* *A_neg* of *None* \Rightarrow *None*
 | *Some (pos_split, neg_split)* \Rightarrow
Some ((pos_split, neg_split), (proj_list_3 L_pos pos_split, proj_list_3 L_neg neg_split)))

lemma *find_sub_False_proj_list_3_same_left*:
assumes *find_sub_False* *xs* *ns* = *Some ((rs, w, ws), (ys, z, zs))*
length xs = *length xs'* *proj_list_3 xs' (rs, w, ws)* = (*rs', w', ws'*)
shows *xs' = rs' @ w' # ws'*
 <proof>

lemma *find_sub_False_proj_list_3_length_left*:
assumes *find_sub_False* *xs* *ns* = *Some ((rs, w, ws), (ys, z, zs))*
length xs = *length xs'* *proj_list_3 xs' (rs, w, ws)* = (*rs', w', ws'*)
shows *length rs* = *length rs'* *length ws* = *length ws'*
 <proof>

lemma *find_sub_False_proj_list_3_same_right*:
assumes *find_sub_False* *xs* *ns* = *Some ((rs, w, ws), (ys, z, zs))*
length ns = *length ns'* *proj_list_3 ns' (ys, z, zs)* = (*ys', z', zs'*)
shows *ns' = ys' @ z' # zs'*
 <proof>

lemma *find_sub_False_proj_list_3_length_right*:
assumes *find_sub_False* *xs* *ns* = *Some ((rs, w, ws), (ys, z, zs))*
length ns = *length ns'* *proj_list_3 ns' (ys, z, zs)* = (*ys', z', zs'*)
shows *length ys* = *length ys'* *length zs* = *length zs'*
 <proof>

lemma *dominate_False_sound*:

assumes *dominate_False* A_pos L_pos A_neg L_neg =
Some (((A_zs , A_x , A_xs), A_ws , A_y , A_ys), ((zs , x , xs), ws , y , ys))
 $length\ A_pos = length\ L_pos$ $length\ A_neg = length\ L_neg$
shows $A_pos = (A_zs @ A_x \# A_xs)$ $A_neg = A_ws @ A_y \# A_ys$
 $L_pos = (zs @ x \# xs)$ $L_neg = ws @ y \# ys$
 $length\ A_ws = length\ ws$ $length\ A_xs = length\ xs$
 $length\ A_ys = length\ ys$ $length\ A_zs = length\ zs$
 $A_y \subseteq A_x$
 <proof>

function *mmulti_join* :: (nat \Rightarrow nat set list \Rightarrow nat set list \Rightarrow 'a table list \Rightarrow 'a table) **where**
mmulti_join n A_pos A_neg L = (if $length\ A_pos + length\ A_neg \neq length\ L$ then {} else
 let $L_pos = take\ (length\ A_pos)\ L$; $L_neg = drop\ (length\ A_pos)\ L$ in
 (case *dominate_True* A_pos L_pos of None \Rightarrow
 (case *dominate_False* A_pos L_pos A_neg L_neg of None \Rightarrow *mmulti_join'* A_pos A_neg L
 | *Some* (((A_zs , A_x , A_xs), A_ws , A_y , A_ys), ((zs , x , xs), ws , y , ys)) \Rightarrow
mmulti_join n ($A_zs @ A_x \# A_xs$) ($A_ws @ A_ys$)
 (($zs @ bin_join\ n\ A_x\ x\ False\ A_y\ y\ \# xs$) @ ($ws @ ys$)))
 | *Some* ((A_zs , A_x , A_xs , A_y , A_ys), (zs , x , xs , y , ys)) \Rightarrow
mmulti_join n ($A_zs @ (A_x \cup A_y) \# A_xs @ A_ys$) A_neg
 (($zs @ bin_join\ n\ A_x\ x\ True\ A_y\ y\ \# xs @ ys$) @ L_neg)))
 <proof>

termination

<proof>

lemma *mmulti_join_link*:

assumes $A_pos \neq []$
and *list_all2* ($\lambda A\ X.$ table $n\ A\ X \wedge wf_set\ n\ A$) ($A_pos @ A_neg$) L
shows *mmulti_join* n A_pos A_neg $L =$ *mmulti_join'* A_pos A_neg L
 <proof>

lemma *mmulti_join_correct*:

assumes $A_pos \neq []$
and *list_all2* ($\lambda A\ X.$ table $n\ A\ X \wedge wf_set\ n\ A$) ($A_pos @ A_neg$) L
shows $z \in$ *mmulti_join* n A_pos A_neg $L \iff wf_tuple\ n\ (\bigcup_{A \in set\ A_pos} A) z \wedge$
 $list_all2\ (\lambda A\ X.$ restrict $A\ z \in X)$ A_pos ($take\ (length\ A_pos)\ L$) \wedge
 $list_all2\ (\lambda A\ X.$ restrict $A\ z \notin X)$ A_neg ($drop\ (length\ A_pos)\ L$)
 <proof>

6 Generic monitoring algorithm

The algorithm defined here abstracts over the implementation of the temporal operators.

6.1 Monitorable formulas

definition *mmonitorable* $\varphi \iff safe_formula\ \varphi \wedge Formula.future_bounded\ \varphi$

definition *mmonitorable_regex* $b\ g\ r \iff safe_regex\ b\ g\ r \wedge Regex.pred_regex\ Formula.future_bounded\ r$

definition *is_simple_eq* :: *Formula.trm* \Rightarrow *Formula.trm* \Rightarrow bool **where**

is_simple_eq $t1\ t2 = (Formula.is_Const\ t1 \wedge (Formula.is_Const\ t2 \vee Formula.is_Var\ t2)) \vee$
 $Formula.is_Var\ t1 \wedge Formula.is_Const\ t2)$

fun *mmonitorable_exec* :: *Formula.formula* \Rightarrow bool **where**

mmonitorable_exec (*Formula.Eq* $t1\ t2$) = *is_simple_eq* $t1\ t2$

$| \text{mmonitorable_exec } (\text{Formula.Neg } (\text{Formula.Eq } (\text{Formula.Var } x) (\text{Formula.Var } y))) = (x = y)$
 $| \text{mmonitorable_exec } (\text{Formula.Pred } e \text{ ts}) = \text{list_all } (\lambda t. \text{Formula.is_Var } t \vee \text{Formula.is_Const } t) \text{ ts}$
 $| \text{mmonitorable_exec } (\text{Formula.Let } p \varphi \psi) = (\{0..<\text{Formula.nfv } \varphi\} \subseteq \text{Formula.fv } \varphi \wedge \text{mmonitorable_exec } \varphi \wedge \text{mmonitorable_exec } \psi)$
 $| \text{mmonitorable_exec } (\text{Formula.Neg } \varphi) = (\text{fv } \varphi = \{\}) \wedge \text{mmonitorable_exec } \varphi$
 $| \text{mmonitorable_exec } (\text{Formula.Or } \varphi \psi) = (\text{fv } \varphi = \text{fv } \psi \wedge \text{mmonitorable_exec } \varphi \wedge \text{mmonitorable_exec } \psi)$
 $| \text{mmonitorable_exec } (\text{Formula.And } \varphi \psi) = (\text{mmonitorable_exec } \varphi \wedge (\text{safe_assignment } (\text{fv } \varphi) \psi \vee \text{mmonitorable_exec } \psi \vee \text{fv } \psi \subseteq \text{fv } \varphi \wedge (\text{is_constraint } \psi \vee (\text{case } \psi \text{ of } \text{Formula.Neg } \psi' \Rightarrow \text{mmonitorable_exec } \psi' \mid _ \Rightarrow \text{False}))))$
 $| \text{mmonitorable_exec } (\text{Formula.Ands } l) = (\text{let } (\text{pos}, \text{neg}) = \text{partition } \text{mmonitorable_exec } l \text{ in } \text{pos} \neq [] \wedge \text{list_all } \text{mmonitorable_exec } (\text{map } \text{remove_neg } \text{neg}) \wedge \bigcup (\text{set } (\text{map } \text{fv } \text{neg})) \subseteq \bigcup (\text{set } (\text{map } \text{fv } \text{pos})))$
 $| \text{mmonitorable_exec } (\text{Formula.Exists } \varphi) = (\text{mmonitorable_exec } \varphi)$
 $| \text{mmonitorable_exec } (\text{Formula.Agg } y \omega \text{ b } f \varphi) = (\text{mmonitorable_exec } \varphi \wedge y + \text{b} \notin \text{Formula.fv } \varphi \wedge \{0..<\text{b}\} \subseteq \text{Formula.fv } \varphi \wedge \text{Formula.fv_trm } f \subseteq \text{Formula.fv } \varphi)$
 $| \text{mmonitorable_exec } (\text{Formula.Prev } I \varphi) = (\text{mmonitorable_exec } \varphi)$
 $| \text{mmonitorable_exec } (\text{Formula.Next } I \varphi) = (\text{mmonitorable_exec } \varphi)$
 $| \text{mmonitorable_exec } (\text{Formula.Since } \varphi I \psi) = (\text{Formula.fv } \varphi \subseteq \text{Formula.fv } \psi \wedge (\text{mmonitorable_exec } \varphi \vee (\text{case } \varphi \text{ of } \text{Formula.Neg } \varphi' \Rightarrow \text{mmonitorable_exec } \varphi' \mid _ \Rightarrow \text{False}))) \wedge \text{mmonitorable_exec } \psi$
 $| \text{mmonitorable_exec } (\text{Formula.Until } \varphi I \psi) = (\text{Formula.fv } \varphi \subseteq \text{Formula.fv } \psi \wedge \text{right } I \neq \infty \wedge (\text{mmonitorable_exec } \varphi \vee (\text{case } \varphi \text{ of } \text{Formula.Neg } \varphi' \Rightarrow \text{mmonitorable_exec } \varphi' \mid _ \Rightarrow \text{False}))) \wedge \text{mmonitorable_exec } \psi$
 $| \text{mmonitorable_exec } (\text{Formula.MatchP } I r) = \text{Regex.safe_regex } \text{Formula.fv } (\lambda g \varphi. \text{mmonitorable_exec } \varphi \vee (g = \text{Lax} \wedge (\text{case } \varphi \text{ of } \text{Formula.Neg } \varphi' \Rightarrow \text{mmonitorable_exec } \varphi' \mid _ \Rightarrow \text{False}))) \text{Past Strict } r$
 $| \text{mmonitorable_exec } (\text{Formula.MatchF } I r) = (\text{Regex.safe_regex } \text{Formula.fv } (\lambda g \varphi. \text{mmonitorable_exec } \varphi \vee (g = \text{Lax} \wedge (\text{case } \varphi \text{ of } \text{Formula.Neg } \varphi' \Rightarrow \text{mmonitorable_exec } \varphi' \mid _ \Rightarrow \text{False})))) \text{Futu Strict } r \wedge \text{right } I \neq \infty$
 $| \text{mmonitorable_exec } _ = \text{False}$

lemma *cases_Neg_iff*:

$(\text{case } \varphi \text{ of } \text{formula.Neg } \psi \Rightarrow P \psi \mid _ \Rightarrow \text{False}) \longleftrightarrow (\exists \psi. \varphi = \text{formula.Neg } \psi \wedge P \psi)$
 $\langle \text{proof} \rangle$

lemma *safe_formula_mmonitorable_exec*: $\text{safe_formula } \varphi \Longrightarrow \text{Formula.future_bounded } \varphi \Longrightarrow \text{mmonitorable_exec } \varphi$

$\langle \text{proof} \rangle$

lemma *safe_assignment_future_bounded*: $\text{safe_assignment } X \varphi \Longrightarrow \text{Formula.future_bounded } \varphi$

$\langle \text{proof} \rangle$

lemma *is_constraint_future_bounded*: $\text{is_constraint } \varphi \Longrightarrow \text{Formula.future_bounded } \varphi$

$\langle \text{proof} \rangle$

lemma *mmonitorable_exec_mmonitorable*: $\text{mmonitorable_exec } \varphi \Longrightarrow \text{mmonitorable } \varphi$

$\langle \text{proof} \rangle$

lemma *monitorable_formula_code*[code]: $\text{mmonitorable } \varphi = \text{mmonitorable_exec } \varphi$

$\langle \text{proof} \rangle$

6.2 Handling regular expressions

datatype *mregex* =

$\text{MSkip } \text{nat}$
 $| \text{MTestPos } \text{nat}$
 $| \text{MTestNeg } \text{nat}$

| *MPlus mregex mregex*
 | *MTimes mregex mregex*
 | *MStar mregex*

primrec ok where

ok _ (*MSkip* *n*) = *True*
 | *ok* *m* (*MTestPos* *n*) = (*n* < *m*)
 | *ok* *m* (*MTestNeg* *n*) = (*n* < *m*)
 | *ok* *m* (*MPlus* *r s*) = (*ok* *m* *r* ∧ *ok* *m* *s*)
 | *ok* *m* (*MTimes* *r s*) = (*ok* *m* *r* ∧ *ok* *m* *s*)
 | *ok* *m* (*MStar* *r*) = *ok* *m* *r*

primrec from_mregex where

from_mregex (*MSkip* *n*) _ = *Regex.Skip* *n*
 | *from_mregex* (*MTestPos* *n*) φ *s* = *Regex.Test* (φ *s* ! *n*)
 | *from_mregex* (*MTestNeg* *n*) φ *s* = (if *safe_formula* (*Formula.Neg* (φ *s* ! *n*))
 then *Regex.Test* (*Formula.Neg* (*Formula.Neg* (*Formula.Neg* (φ *s* ! *n*))))
 else *Regex.Test* (*Formula.Neg* (φ *s* ! *n*)))
 | *from_mregex* (*MPlus* *r s*) φ *s* = *Regex.Plus* (*from_mregex* *r* φ *s*) (*from_mregex* *s* φ *s*)
 | *from_mregex* (*MTimes* *r s*) φ *s* = *Regex.Times* (*from_mregex* *r* φ *s*) (*from_mregex* *s* φ *s*)
 | *from_mregex* (*MStar* *r*) φ *s* = *Regex.Star* (*from_mregex* *r* φ *s*)

primrec to_mregex_exec where

to_mregex_exec (*Regex.Skip* *n*) *xs* = (*MSkip* *n*, *xs*)
 | *to_mregex_exec* (*Regex.Test* φ) *xs* = (if *safe_formula* φ then (*MTestPos* (*length* *xs*), *xs* @ [φ])
 else case φ of *Formula.Neg* φ' \Rightarrow (*MTestNeg* (*length* *xs*), *xs* @ [φ']) | _ \Rightarrow (*MSkip* 0, *xs*))
 | *to_mregex_exec* (*Regex.Plus* *r s*) *xs* =
 (let (*mr*, *ys*) = *to_mregex_exec* *r* *xs*; (*ms*, *zs*) = *to_mregex_exec* *s* *ys*
 in (*MPlus* *mr* *ms*, *zs*))
 | *to_mregex_exec* (*Regex.Times* *r s*) *xs* =
 (let (*mr*, *ys*) = *to_mregex_exec* *r* *xs*; (*ms*, *zs*) = *to_mregex_exec* *s* *ys*
 in (*MTimes* *mr* *ms*, *zs*))
 | *to_mregex_exec* (*Regex.Star* *r*) *xs* =
 (let (*mr*, *ys*) = *to_mregex_exec* *r* *xs* in (*MStar* *mr*, *ys*))

primrec shift where

shift (*MSkip* *n*) *k* = *MSkip* *n*
 | *shift* (*MTestPos* *i*) *k* = *MTestPos* (*i* + *k*)
 | *shift* (*MTestNeg* *i*) *k* = *MTestNeg* (*i* + *k*)
 | *shift* (*MPlus* *r s*) *k* = *MPlus* (*shift* *r* *k*) (*shift* *s* *k*)
 | *shift* (*MTimes* *r s*) *k* = *MTimes* (*shift* *r* *k*) (*shift* *s* *k*)
 | *shift* (*MStar* *r*) *k* = *MStar* (*shift* *r* *k*)

primrec to_mregex where

to_mregex (*Regex.Skip* *n*) = (*MSkip* *n*, [])
 | *to_mregex* (*Regex.Test* φ) = (if *safe_formula* φ then (*MTestPos* 0, [φ])
 else case φ of *Formula.Neg* φ' \Rightarrow (*MTestNeg* 0, [φ']) | _ \Rightarrow (*MSkip* 0, []))
 | *to_mregex* (*Regex.Plus* *r s*) =
 (let (*mr*, *ys*) = *to_mregex* *r*; (*ms*, *zs*) = *to_mregex* *s*
 in (*MPlus* *mr* (*shift* *ms* (*length* *ys*)), *ys* @ *zs*))
 | *to_mregex* (*Regex.Times* *r s*) =
 (let (*mr*, *ys*) = *to_mregex* *r*; (*ms*, *zs*) = *to_mregex* *s*
 in (*MTimes* *mr* (*shift* *ms* (*length* *ys*)), *ys* @ *zs*))
 | *to_mregex* (*Regex.Star* *r*) =
 (let (*mr*, *ys*) = *to_mregex* *r* in (*MStar* *mr*, *ys*))

lemma *shift_0*: *shift* *r* 0 = *r*
 <proof>

lemma *shift_shift*: $\text{shift} (\text{shift } r \ k) \ j = \text{shift } r \ (k + j)$
 ⟨proof⟩

lemma *to_mregex_to_mregex_exec*:
 $\text{case } \text{to_mregex } r \ \text{of } (mr, \varphi s) \Rightarrow \text{to_mregex_exec } r \ xs = (\text{shift } mr \ (\text{length } xs), xs \ @ \ \varphi s)$
 ⟨proof⟩

lemma *to_mregex_to_mregex_exec_Nil*[code]: $\text{to_mregex } r = \text{to_mregex_exec } r \ []$
 ⟨proof⟩

lemma *ok_mono*: $\text{ok } m \ mr \Longrightarrow m \leq n \Longrightarrow \text{ok } n \ mr$
 ⟨proof⟩

lemma *from_mregex_cong*: $\text{ok } m \ mr \Longrightarrow (\forall i < m. xs \ ! \ i = ys \ ! \ i) \Longrightarrow \text{from_mregex } mr \ xs = \text{from_mregex } mr \ ys$
 ⟨proof⟩

lemma *not_Neg_cases*:
 $(\forall \psi. \varphi \neq \text{Formula.Neg } \psi) \Longrightarrow (\text{case } \varphi \ \text{of } \text{formula.Neg } \psi \Rightarrow f \ \psi \ | \ _ \Rightarrow x) = x$
 ⟨proof⟩

lemma *to_mregex_exec_ok*:
 $\text{to_mregex_exec } r \ xs = (mr, ys) \Longrightarrow \exists zs. ys = xs \ @ \ zs \wedge \text{set } zs = \text{atms } r \wedge \text{ok } (\text{length } ys) \ mr$
 ⟨proof⟩

lemma *ok_shift*: $\text{ok } (i + m) \ (\text{Monitor.shift } r \ i) \longleftrightarrow \text{ok } m \ r$
 ⟨proof⟩

lemma *to_mregex_ok*: $\text{to_mregex } r = (mr, ys) \Longrightarrow \text{set } ys = \text{atms } r \wedge \text{ok } (\text{length } ys) \ mr$
 ⟨proof⟩

lemma *from_mregex_shift*: $\text{from_mregex } (\text{shift } r \ (\text{length } xs)) \ (xs \ @ \ ys) = \text{from_mregex } r \ ys$
 ⟨proof⟩

lemma *from_mregex_to_mregex*: $\text{safe_regex } m \ g \ r \Longrightarrow \text{case_prod } \text{from_mregex} \ (\text{to_mregex } r) = r$
 ⟨proof⟩

lemma *from_mregex_eq*: $\text{safe_regex } m \ g \ r \Longrightarrow \text{to_mregex } r = (mr, \varphi s) \Longrightarrow \text{from_mregex } mr \ \varphi s = r$
 ⟨proof⟩

lemma *from_mregex_to_mregex_exec*: $\text{safe_regex } m \ g \ r \Longrightarrow \text{case_prod } \text{from_mregex} \ (\text{to_mregex_exec } r \ xs) = r$
 ⟨proof⟩

derive *linorder mregex*

6.2.1 LPD

definition *saturate where*
 $\text{saturate } f = \text{while } (\lambda S. f \ S \neq S) \ f$

lemma *saturate_code*[code]:
 $\text{saturate } f \ S = (\text{let } S' = f \ S \ \text{in } \text{if } S' = S \ \text{then } S \ \text{else } \text{saturate } f \ S')$
 ⟨proof⟩

definition *MTimesL* $r \ S = \text{MTimes } r \ ' \ S$

definition *MTimesR* $R \ s = (\lambda r. \text{MTimes } r \ s) \ ' \ R$

primrec *LPD* **where**

$LPD (MSkip\ n) = (case\ n\ of\ 0 \Rightarrow \{\} \mid Suc\ m \Rightarrow \{MSkip\ m\})$
 $| LPD (MTestPos\ \varphi) = \{\}$
 $| LPD (MTestNeg\ \varphi) = \{\}$
 $| LPD (MPlus\ r\ s) = (LPD\ r \cup LPD\ s)$
 $| LPD (MTimes\ r\ s) = MTimesR (LPD\ r)\ s \cup LPD\ s$
 $| LPD (MStar\ r) = MTimesR (LPD\ r)\ (MStar\ r)$

primrec *LPDi* **where**

$LPDi\ 0\ r = \{r\}$
 $| LPDi\ (Suc\ i)\ r = (\bigcup s \in LPD\ r.\ LPDi\ i\ s)$

lemma *LPDi_Suc_alt*: $LPDi\ (Suc\ i)\ r = (\bigcup s \in LPDi\ i\ r.\ LPD\ s)$
 $\langle proof \rangle$

definition *LPDs* $r = (\bigcup i.\ LPDi\ i\ r)$

lemma *LPDs_refl*: $r \in LPDs\ r$
 $\langle proof \rangle$

lemma *LPDs_trans*: $r \in LPD\ s \implies s \in LPDs\ t \implies r \in LPDs\ t$
 $\langle proof \rangle$

lemma *LPDi_Test*:

$LPDi\ i\ (MSkip\ 0) \subseteq \{MSkip\ 0\}$
 $LPDi\ i\ (MTestPos\ \varphi) \subseteq \{MTestPos\ \varphi\}$
 $LPDi\ i\ (MTestNeg\ \varphi) \subseteq \{MTestNeg\ \varphi\}$
 $\langle proof \rangle$

lemma *LPDs_Test*:

$LPDs\ (MSkip\ 0) \subseteq \{MSkip\ 0\}$
 $LPDs\ (MTestPos\ \varphi) \subseteq \{MTestPos\ \varphi\}$
 $LPDs\ (MTestNeg\ \varphi) \subseteq \{MTestNeg\ \varphi\}$
 $\langle proof \rangle$

lemma *LPDi_MSkip*: $LPDi\ i\ (MSkip\ n) \subseteq MSkip\ \{i.\ i \leq n\}$
 $\langle proof \rangle$

lemma *LPDs_MSkip*: $LPDs\ (MSkip\ n) \subseteq MSkip\ \{i.\ i \leq n\}$
 $\langle proof \rangle$

lemma *LPDi_Plus*: $LPDi\ i\ (MPlus\ r\ s) \subseteq \{MPlus\ r\ s\} \cup LPDi\ i\ r \cup LPDi\ i\ s$
 $\langle proof \rangle$

lemma *LPDs_Plus*: $LPDs\ (MPlus\ r\ s) \subseteq \{MPlus\ r\ s\} \cup LPDs\ r \cup LPDs\ s$
 $\langle proof \rangle$

lemma *LPDi_Times*:

$LPDi\ i\ (MTimes\ r\ s) \subseteq \{MTimes\ r\ s\} \cup MTimesR (\bigcup j \leq i.\ LPDi\ j\ r)\ s \cup (\bigcup j \leq i.\ LPDi\ j\ s)$
 $\langle proof \rangle$

lemma *LPDs_Times*: $LPDs\ (MTimes\ r\ s) \subseteq \{MTimes\ r\ s\} \cup MTimesR (LPDs\ r)\ s \cup LPDs\ s$
 $\langle proof \rangle$

lemma *LPDi_Star*: $j \leq i \implies LPDi\ j\ (MStar\ r) \subseteq \{MStar\ r\} \cup MTimesR (\bigcup j \leq i.\ LPDi\ j\ r)\ (MStar\ r)$
 $\langle proof \rangle$

lemma *LPDs_Star*: $LPDs (MStar r) \subseteq \{MStar r\} \cup MTimesR (LPDs r) (MStar r)$
 ⟨proof⟩

lemma *finite_LPDS*: $finite (LPDs r)$
 ⟨proof⟩

context begin

private abbreviation (*input*) $addLPD r \equiv \lambda S. insert r S \cup Set.bind (insert r S) LPD$

private lemma *mono_addLPD*: $mono (addLPD r)$
 ⟨proof⟩ **lemma** *LPDs_aux1*: $lfp (addLPD r) \subseteq LPDs r$
 ⟨proof⟩ **lemma** *LPDs_aux2*: $LPDi i r \subseteq lfp (addLPD r)$
 ⟨proof⟩

lemma *LPDs_alt*: $LPDs r = lfp (addLPD r)$
 ⟨proof⟩

lemma *LPDs_code*[*code*]:
 $LPDs r = saturate (addLPD r) \{\}$
 ⟨proof⟩

end

6.2.2 RPD

primrec *RPD* **where**
 $RPD (MSkip n) = (case\ n\ of\ 0 \Rightarrow \{\} \mid Suc\ m \Rightarrow \{MSkip\ m\})$
 $RPD (MTestPos\ \varphi) = \{\}$
 $RPD (MTestNeg\ \varphi) = \{\}$
 $RPD (MPlus\ r\ s) = (RPD\ r \cup RPD\ s)$
 $RPD (MTimes\ r\ s) = MTimesL\ r\ (RPD\ s) \cup RPD\ r$
 $RPD (MStar\ r) = MTimesL\ (MStar\ r)\ (RPD\ r)$

primrec *RPDi* **where**
 $RPDi\ 0\ r = \{r\}$
 $RPDi\ (Suc\ i)\ r = (\bigcup s \in RPD\ r. RPDi\ i\ s)$

lemma *RPDi_Suc_alt*: $RPDi\ (Suc\ i)\ r = (\bigcup s \in RPDi\ i\ r. RPD\ s)$
 ⟨proof⟩

definition *RPDs* $r = (\bigcup i. RPDi\ i\ r)$

lemma *RPDs_refl*: $r \in RPDs\ r$
 ⟨proof⟩

lemma *RPDs_trans*: $r \in RPD\ s \implies s \in RPDs\ t \implies r \in RPDs\ t$
 ⟨proof⟩

lemma *RPDi_Test*:
 $RPDi\ i\ (MSkip\ 0) \subseteq \{MSkip\ 0\}$
 $RPDi\ i\ (MTestPos\ \varphi) \subseteq \{MTestPos\ \varphi\}$
 $RPDi\ i\ (MTestNeg\ \varphi) \subseteq \{MTestNeg\ \varphi\}$
 ⟨proof⟩

lemma *RPDs_Test*:
 $RPDs\ (MSkip\ 0) \subseteq \{MSkip\ 0\}$
 $RPDs\ (MTestPos\ \varphi) \subseteq \{MTestPos\ \varphi\}$
 $RPDs\ (MTestNeg\ \varphi) \subseteq \{MTestNeg\ \varphi\}$
 ⟨proof⟩

lemma *RPDi_MSkip*: $RPDi\ i\ (MSkip\ n) \subseteq MSkip\ \{i.\ i \leq n\}$
 ⟨proof⟩

lemma *RPDs_MSkip*: $RPDs\ (MSkip\ n) \subseteq MSkip\ \{i.\ i \leq n\}$
 ⟨proof⟩

lemma *RPDi_Plus*: $RPDi\ i\ (MPlus\ r\ s) \subseteq \{MPlus\ r\ s\} \cup RPDi\ i\ r \cup RPDi\ i\ s$
 ⟨proof⟩

lemma *RPDi_Suc_RPD_Plus*:
 $RPDi\ (Suc\ i)\ r \subseteq RPDs\ (MPlus\ r\ s)$
 $RPDi\ (Suc\ i)\ s \subseteq RPDs\ (MPlus\ r\ s)$
 ⟨proof⟩

lemma *RPDs_Plus*: $RPDs\ (MPlus\ r\ s) \subseteq \{MPlus\ r\ s\} \cup RPDs\ r \cup RPDs\ s$
 ⟨proof⟩

lemma *RPDi_Times*:
 $RPDi\ i\ (MTimes\ r\ s) \subseteq \{MTimes\ r\ s\} \cup MTimesL\ r\ (\bigcup_{j \leq i} RPDi\ j\ s) \cup (\bigcup_{j \leq i} RPDi\ j\ r)$
 ⟨proof⟩

lemma *RPDs_Times*: $RPDs\ (MTimes\ r\ s) \subseteq \{MTimes\ r\ s\} \cup MTimesL\ r\ (RPDs\ s) \cup RPDs\ r$
 ⟨proof⟩

lemma *RPDi_Star*: $j \leq i \implies RPDi\ j\ (MStar\ r) \subseteq \{MStar\ r\} \cup MTimesL\ (MStar\ r)\ (\bigcup_{j \leq i} RPDi\ j\ r)$
 ⟨proof⟩

lemma *RPDs_Star*: $RPDs\ (MStar\ r) \subseteq \{MStar\ r\} \cup MTimesL\ (MStar\ r)\ (RPDs\ r)$
 ⟨proof⟩

lemma *finite_RPDs*: $finite\ (RPDs\ r)$
 ⟨proof⟩

context begin

private abbreviation *addRPD* $r \equiv \lambda S. insert\ r\ S \cup Set.bind\ (insert\ r\ S)\ RPD$

private lemma *mono_addRPD*: $mono\ (addRPD\ r)$
 ⟨proof⟩ **lemma** *RPDs_aux1*: $lfp\ (addRPD\ r) \subseteq RPDs\ r$
 ⟨proof⟩ **lemma** *RPDs_aux2*: $RPDi\ i\ r \subseteq lfp\ (addRPD\ r)$
 ⟨proof⟩

lemma *RPDs_alt*: $RPDs\ r = lfp\ (addRPD\ r)$
 ⟨proof⟩

lemma *RPDs_code*[code]:
 $RPDs\ r = saturate\ (addRPD\ r)\ \{\}$
 ⟨proof⟩

end

6.3 The executable monitor

type_synonym *ts* = *nat*

type_synonym *'a mbuf2* = *'a table list* × *'a table list*

```

type_synonym 'a mbufn = 'a table list list
type_synonym 'a msaux = (ts × 'a table) list
type_synonym 'a muaux = (ts × 'a table × 'a table) list
type_synonym 'a mrdaux = (ts × (mregex, 'a table) mapping) list
type_synonym 'a mlδaux = (ts × 'a table list × 'a table) list

```

```

datatype mconstraint = MEq | MLess | MLessEq

```

```

record args =
  args_ivl ::  $\mathcal{I}$ 
  args_n :: nat
  args_L :: nat set
  args_R :: nat set
  args_pos :: bool

```

```

datatype (dead 'msaux, dead 'muaux) mformula =
  MRel event_data table
  | MPred Formula.name Formula.trm list
  | MLet Formula.name nat ('msaux, 'muaux) mformula ('msaux, 'muaux) mformula
  | MAnd nat set ('msaux, 'muaux) mformula bool nat set ('msaux, 'muaux) mformula event_data mbuf2
  | MAndAssign ('msaux, 'muaux) mformula nat × Formula.trm
  | MAndRel ('msaux, 'muaux) mformula Formula.trm × bool × mconstraint × Formula.trm
  | MAnds nat set list nat set list ('msaux, 'muaux) mformula list event_data mbufn
  | MOr ('msaux, 'muaux) mformula ('msaux, 'muaux) mformula event_data mbuf2
  | MNeg ('msaux, 'muaux) mformula
  | MExists ('msaux, 'muaux) mformula
  | MAgg bool nat Formula.agg_op nat Formula.trm ('msaux, 'muaux) mformula
  | MPrev  $\mathcal{I}$  ('msaux, 'muaux) mformula bool event_data table list ts list
  | MNext  $\mathcal{I}$  ('msaux, 'muaux) mformula bool ts list
  | MSince args ('msaux, 'muaux) mformula ('msaux, 'muaux) mformula event_data mbuf2 ts list 'msaux
  | MUntil args ('msaux, 'muaux) mformula ('msaux, 'muaux) mformula event_data mbuf2 ts list 'muaux
  | MMatchP  $\mathcal{I}$  mregex mregex list ('msaux, 'muaux) mformula list event_data mbufn ts list event_data
mrdaux
  | MMatchF  $\mathcal{I}$  mregex mregex list ('msaux, 'muaux) mformula list event_data mbufn ts list event_data
mlδaux

```

```

record ('msaux, 'muaux) mstate =
  mstate_i :: nat
  mstate_m :: ('msaux, 'muaux) mformula
  mstate_n :: nat

```

```

fun eq_rel :: nat ⇒ Formula.trm ⇒ Formula.trm ⇒ event_data table where
  eq_rel n (Formula.Const x) (Formula.Const y) = (if x = y then unit_table n else empty_table)
  | eq_rel n (Formula.Var x) (Formula.Const y) = singleton_table n x y
  | eq_rel n (Formula.Const x) (Formula.Var y) = singleton_table n y x
  | eq_rel n _ _ = undefined

```

```

lemma regex_atms_size:  $x \in \text{regex.atms } r \implies \text{size } x < \text{regex.size\_regex size } r$ 
  ⟨proof⟩

```

```

lemma atms_size:
  assumes  $x \in \text{atms } r$ 
  shows  $\text{size } x < \text{Regex.size\_regex size } r$ 
  ⟨proof⟩

```

```

definition init_args ::  $\mathcal{I} \Rightarrow \text{nat} \Rightarrow \text{nat set} \Rightarrow \text{nat set} \Rightarrow \text{bool} \Rightarrow \text{args}$  where
  init_args I n L R pos = (args_ivl = I, args_n = n, args_L = L, args_R = R, args_pos = pos)

```

```

locale msaux =
  fixes valid_msaux :: args ⇒ ts ⇒ 'msaux ⇒ event_data msaux ⇒ bool
    and init_msaux :: args ⇒ 'msaux
    and add_new_ts_msaux :: args ⇒ ts ⇒ 'msaux ⇒ 'msaux
    and join_msaux :: args ⇒ event_data table ⇒ 'msaux ⇒ 'msaux
    and add_new_table_msaux :: args ⇒ event_data table ⇒ 'msaux ⇒ 'msaux
    and result_msaux :: args ⇒ 'msaux ⇒ event_data table
  assumes valid_init_msaux: L ⊆ R ⇒
    valid_msaux (init_args I n L R pos) 0 (init_msaux (init_args I n L R pos)) []
  assumes valid_add_new_ts_msaux: valid_msaux args cur aux auxlist ⇒ nt ≥ cur ⇒
    valid_msaux args nt (add_new_ts_msaux args nt aux)
    (filter (λ(t, rel). enat (nt - t) ≤ right (args_ivl args)) auxlist)
  assumes valid_join_msaux: valid_msaux args cur aux auxlist ⇒
    table (args_n args) (args_L args) rel1 ⇒
    valid_msaux args cur (join_msaux args rel1 aux)
    (map (λ(t, rel). (t, join rel (args_pos args) rel1)) auxlist)
  assumes valid_add_new_table_msaux: valid_msaux args cur aux auxlist ⇒
    table (args_n args) (args_R args) rel2 ⇒
    valid_msaux args cur (add_new_table_msaux args rel2 aux)
    (case auxlist of
      [] => [(cur, rel2)]
    | ((t, y) # ts) => if t = cur then (t, y ∪ rel2) # ts else (cur, rel2) # auxlist)
  and valid_result_msaux: valid_msaux args cur aux auxlist ⇒ result_msaux args aux =
    foldr (∪) [rel. (t, rel) ← auxlist, left (args_ivl args) ≤ cur - t] {}

fun check_before :: I ⇒ ts ⇒ (ts × 'a × 'b) ⇒ bool where
  check_before I dt (t, a, b) ⇔ enat t + right I < enat dt

fun proj_thd :: ('a × 'b × 'c) ⇒ 'c where
  proj_thd (t, a1, a2) = a2

definition update_until :: args ⇒ event_data table ⇒ event_data table ⇒ ts ⇒ event_data muaux ⇒
  event_data muaux where
  update_until args rel1 rel2 nt aux =
    (map (λx. case x of (t, a1, a2) ⇒ (t, if (args_pos args) then join a1 True rel1 else a1 ∪ rel1,
      if mem (nt - t) (args_ivl args) then a2 ∪ join rel2 (args_pos args) a1 else a2)) aux) @
    [(nt, rel1, if left (args_ivl args) = 0 then rel2 else empty_table)]

lemma map_proj_thd_update_until: map proj_thd (takeWhile (check_before (args_ivl args) nt) auxlist)
  =
  map proj_thd (takeWhile (check_before (args_ivl args) nt) (update_until args rel1 rel2 nt auxlist))
  ⟨proof⟩

fun eval_until :: I ⇒ ts ⇒ event_data muaux ⇒ event_data table list × event_data 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))

lemma eval_until_length:
  eval_until I nt auxlist = (res, auxlist') ⇒ length auxlist = length res + length auxlist'
  ⟨proof⟩

lemma eval_until_res: eval_until I nt auxlist = (res, auxlist') ⇒
  res = map proj_thd (takeWhile (check_before I nt) auxlist)
  ⟨proof⟩

lemma eval_until_auxlist': eval_until I nt auxlist = (res, auxlist') ⇒
  auxlist' = drop (length res) auxlist

```

<proof>

locale *muaux* =

```
fixes valid_muaux :: args ⇒ ts ⇒ 'muaux ⇒ event_data muaux ⇒ bool
and init_muaux :: args ⇒ 'muaux
and add_new_muaux :: args ⇒ event_data table ⇒ event_data table ⇒ ts ⇒ 'muaux ⇒ 'muaux
and length_muaux :: args ⇒ 'muaux ⇒ nat
and eval_muaux :: args ⇒ ts ⇒ 'muaux ⇒ event_data table list × 'muaux
assumes valid_init_muaux: L ⊆ R ⇒
  valid_muaux (init_args I n L R pos) 0 (init_muaux (init_args I n L R pos)) []
assumes valid_add_new_muaux: valid_muaux args cur aux auxlist ⇒
  table (args_n args) (args_L args) rel1 ⇒
  table (args_n args) (args_R args) rel2 ⇒
  nt ≥ cur ⇒
  valid_muaux args nt (add_new_muaux args rel1 rel2 nt aux)
  (update_until args rel1 rel2 nt auxlist)
assumes valid_length_muaux: valid_muaux args cur aux auxlist ⇒ length_muaux args aux = length
auxlist
assumes valid_eval_muaux: valid_muaux args cur aux auxlist ⇒ nt ≥ cur ⇒
  eval_muaux args nt aux = (res, aux') ⇒ eval_until (args_ivl args) nt auxlist = (res', auxlist') ⇒
  res = res' ∧ valid_muaux args cur aux' auxlist'
```

locale *maux* = *msaux* *valid_msaux* *init_msaux* *add_new_ts_msaux* *join_msaux* *add_new_table_msaux* *result_msaux* +

```
muaux valid_muaux init_muaux add_new_muaux length_muaux eval_muaux
for valid_msaux :: args ⇒ ts ⇒ 'msaux ⇒ event_data msaux ⇒ bool
and init_msaux :: args ⇒ 'msaux
and add_new_ts_msaux :: args ⇒ ts ⇒ 'msaux ⇒ 'msaux
and join_msaux :: args ⇒ event_data table ⇒ 'msaux ⇒ 'msaux
and add_new_table_msaux :: args ⇒ event_data table ⇒ 'msaux ⇒ 'msaux
and result_msaux :: args ⇒ 'msaux ⇒ event_data table
and valid_muaux :: args ⇒ ts ⇒ 'muaux ⇒ event_data muaux ⇒ bool
and init_muaux :: args ⇒ 'muaux
and add_new_muaux :: args ⇒ event_data table ⇒ event_data table ⇒ ts ⇒ 'muaux ⇒ 'muaux
and length_muaux :: args ⇒ 'muaux ⇒ nat
and eval_muaux :: args ⇒ nat ⇒ 'muaux ⇒ event_data table list × 'muaux
```

fun *split_assignment* :: nat set ⇒ Formula.formula ⇒ nat × Formula.trm **where**

```
split_assignment X (Formula.Eq t1 t2) = (case (t1, t2) of
  (Formula.Var x, Formula.Var y) ⇒ if x ∈ X then (y, t1) else (x, t2)
| (Formula.Var x, _) ⇒ (x, t2)
| (_, Formula.Var y) ⇒ (y, t1))
| split_assignment _ _ = undefined
```

fun *split_constraint* :: Formula.formula ⇒ Formula.trm × bool × mconstraint × Formula.trm **where**

```
split_constraint (Formula.Eq t1 t2) = (t1, True, MEq, t2)
| split_constraint (Formula.Less t1 t2) = (t1, True, MLess, t2)
| split_constraint (Formula.LessEq t1 t2) = (t1, True, MLessEq, t2)
| split_constraint (Formula.Neg (Formula.Eq t1 t2)) = (t1, False, MEq, t2)
| split_constraint (Formula.Neg (Formula.Less t1 t2)) = (t1, False, MLess, t2)
| split_constraint (Formula.Neg (Formula.LessEq t1 t2)) = (t1, False, MLessEq, t2)
| split_constraint _ = undefined
```

function (**in** *maux*) (*sequential*) *minit0* :: nat ⇒ Formula.formula ⇒ ('msaux, 'muaux) mformula **where**

```
minit0 n (Formula.Neg φ) = (if fv φ = {} then MNeg (minit0 n φ) else MRel empty_table)
| minit0 n (Formula.Eq t1 t2) = MRel (eq_rel n t1 t2)
| minit0 n (Formula.Pred e ts) = MPred e ts
| minit0 n (Formula.Let p φ ψ) = MLet p (Formula.nfv φ) (minit0 (Formula.nfv φ) φ) (minit0 n ψ)
```

```

| minit0 n (Formula.Or  $\varphi$   $\psi$ ) = MOr (minit0 n  $\varphi$ ) (minit0 n  $\psi$ ) ([], [])
| minit0 n (Formula.And  $\varphi$   $\psi$ ) = (if safe_assignment (fv  $\varphi$ )  $\psi$  then
  MAndAssign (minit0 n  $\varphi$ ) (split_assignment (fv  $\varphi$ )  $\psi$ )
  else if safe_formula  $\psi$  then
    MAnd (fv  $\varphi$ ) (minit0 n  $\varphi$ ) True (fv  $\psi$ ) (minit0 n  $\psi$ ) ([], [])
  else if is_constraint  $\psi$  then
    MAndRel (minit0 n  $\varphi$ ) (split_constraint  $\psi$ )
  else (case  $\psi$  of Formula.Neg  $\psi \Rightarrow$ 
    MAnd (fv  $\varphi$ ) (minit0 n  $\varphi$ ) False (fv  $\psi$ ) (minit0 n  $\psi$ ) ([], []))
| minit0 n (Formula.Ands l) = (let (pos, neg) = partition_safe_formula l in
  let mpos = map (minit0 n) pos in
  let mneg = map (minit0 n) (map remove_neg neg) in
  let vpos = map fv pos in
  let vneg = map fv neg in
  MAnds vpos vneg (mpos @ mneg) (replicate (length l) []))
| minit0 n (Formula.Exists  $\varphi$ ) = MExists (minit0 (Suc n)  $\varphi$ )
| minit0 n (Formula.Agg y  $\omega$  b f  $\varphi$ ) = MAgg (fv  $\varphi \subseteq \{0..<b\}$ ) y  $\omega$  b f (minit0 (b + n)  $\varphi$ )
| minit0 n (Formula.Prev I  $\varphi$ ) = MPrev I (minit0 n  $\varphi$ ) True [] []
| minit0 n (Formula.Next I  $\varphi$ ) = MNext I (minit0 n  $\varphi$ ) True [] []
| minit0 n (Formula.Since  $\varphi$  I  $\psi$ ) = (if safe_formula  $\varphi$ 
  then MSince (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) True) (minit0 n  $\varphi$ ) (minit0 n  $\psi$ ) ([], []) []
  (init_msaux (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) True))
  else (case  $\varphi$  of
    Formula.Neg  $\varphi \Rightarrow$  MSince (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) False) (minit0 n  $\varphi$ ) (minit0
n  $\psi$ ) ([], []) [] (init_msaux (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) False))
    | _  $\Rightarrow$  undefined))
| minit0 n (Formula.Until  $\varphi$  I  $\psi$ ) = (if safe_formula  $\varphi$ 
  then MUntil (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) True) (minit0 n  $\varphi$ ) (minit0 n  $\psi$ ) ([], []) []
  (init_muaux (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) True))
  else (case  $\varphi$  of
    Formula.Neg  $\varphi \Rightarrow$  MUntil (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) False) (minit0 n  $\varphi$ ) (minit0
n  $\psi$ ) ([], []) [] (init_muaux (init_args I n (Formula.fv  $\varphi$ ) (Formula.fv  $\psi$ ) False))
    | _  $\Rightarrow$  undefined))
| minit0 n (Formula.MatchP I r) =
  (let (mr,  $\varphi$ s) = to_mregex r
  in MMatchP I mr (sorted_list_of_set (RPDs mr)) (map (minit0 n)  $\varphi$ s) (replicate (length  $\varphi$ s) []) [] [])
| minit0 n (Formula.MatchF I r) =
  (let (mr,  $\varphi$ s) = to_mregex r
  in MMatchF I mr (sorted_list_of_set (LPDs mr)) (map (minit0 n)  $\varphi$ s) (replicate (length  $\varphi$ s) []) [] [])
| minit0 n _ = undefined
<proof>
termination (in maux)
<proof>

```

definition (in *maux*) *minit* :: *Formula.formula* \Rightarrow (*'msaux*, *'muaux*) *mstate* **where**
minit φ = (let *n* = *Formula.nfv* φ in (*mstate_i* = 0, *mstate_m* = *minit0* *n* φ , *mstate_n* = *n*))

definition (in *maux*) *minit_safe* **where**
minit_safe φ = (if *mmonitorable_exec* φ then *minit* φ else *undefined*)

fun *mprev_next* :: $\mathcal{I} \Rightarrow$ *event_data table list* \Rightarrow *ts list* \Rightarrow *event_data table list* \times *event_data table list* \times *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* :: *event_data table list* \Rightarrow *event_data table list* \Rightarrow *event_data mbuf2* \Rightarrow *event_data mbuf2* **where**
mbuf2_add *xs' ys'* (*xs*, *ys*) = (*xs* @ *xs'*, *ys* @ *ys'*)

fun *mbuf2_take* :: (*event_data table* \Rightarrow *event_data table* \Rightarrow 'b) \Rightarrow *event_data mbuf2* \Rightarrow 'b *list* \times *event_data 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* :: (*event_data table* \Rightarrow *event_data table* \Rightarrow *ts* \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow *event_data mbuf2* \Rightarrow *ts list* \Rightarrow 'b \times *event_data mbuf2* \times *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*)

lemma *size_list_length_diff1*: *xs* \neq [] \implies [] \notin *set xs* \implies
size_list (λ *xs*. *length xs* - *Suc* 0) *xs* < *size_list length xs*
<proof>

fun *mbufn_add* :: *event_data table list list* \Rightarrow *event_data mbufn* \Rightarrow *event_data mbufn* **where**
mbufn_add *xs' xs* = *List.map2* (@) *xs xs'*

function *mbufn_take* :: (*event_data table list* \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow *event_data mbufn* \Rightarrow 'b \times *event_data mbufn* **where**
mbufn_take *f* *z* *buf* = (if *buf* = [] \vee [] \in *set buf* then (*z*, *buf*)
else *mbufn_take* *f* (*f* (*map hd buf*) *z*) (*map tl buf*))
<proof>

termination <proof>

fun *mbufnt_take* :: (*event_data table list* \Rightarrow *ts* \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow *event_data mbufn* \Rightarrow *ts list* \Rightarrow 'b \times *event_data mbufn* \times *ts list* **where**
mbufnt_take *f* *z* *buf* *ts* =
(if [] \in *set buf* \vee *ts* = [] then (*z*, *buf*, *ts*)
else *mbufnt_take* *f* (*f* (*map hd buf*) (*hd ts*) *z*) (*map tl buf*) (*tl ts*))

fun *match* :: *Formula.trm list* \Rightarrow *event_data list* \Rightarrow (*nat* \rightarrow *event_data*) *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 \Rightarrow *None*
| *Some f* \Rightarrow (case *f* *x* of
None \Rightarrow *Some* (*f*(*x* \mapsto *y*))
| *Some z* \Rightarrow if *y* = *z* then *Some f* else *None*))
| *match* _ _ = *None*

fun *meval_trm* :: *Formula.trm* \Rightarrow *event_data tuple* \Rightarrow *event_data* **where**
meval_trm (*Formula.Var* *x*) *v* = *the* (*v* ! *x*)
| *meval_trm* (*Formula.Const* *x*) *v* = *x*
| *meval_trm* (*Formula.Plus* *x* *y*) *v* = *meval_trm* *x* *v* + *meval_trm* *y* *v*
| *meval_trm* (*Formula.Minus* *x* *y*) *v* = *meval_trm* *x* *v* - *meval_trm* *y* *v*
| *meval_trm* (*Formula.UMinus* *x*) *v* = - *meval_trm* *x* *v*
| *meval_trm* (*Formula.Mult* *x* *y*) *v* = *meval_trm* *x* *v* * *meval_trm* *y* *v*
| *meval_trm* (*Formula.Div* *x* *y*) *v* = *meval_trm* *x* *v* div *meval_trm* *y* *v*
| *meval_trm* (*Formula.Mod* *x* *y*) *v* = *meval_trm* *x* *v* mod *meval_trm* *y* *v*
| *meval_trm* (*Formula.F2i* *x*) *v* = *EInt* (*integer_of_event_data* (*meval_trm* *x* *v*))
| *meval_trm* (*Formula.I2f* *x*) *v* = *EFloat* (*double_of_event_data* (*meval_trm* *x* *v*))

definition *eval_agg* :: *nat* \Rightarrow *bool* \Rightarrow *nat* \Rightarrow *Formula.agg_op* \Rightarrow *nat* \Rightarrow *Formula.trm* \Rightarrow *event_data table* \Rightarrow *event_data table* **where**

```

eval_agg n g0 y ω b f rel = (if g0 ∧ rel = empty_table
  then singleton_table n y (eval_agg_op ω { })
  else (λk.
    let group = Set.filter (λx. drop b x = k) rel;
    M = (λy. (y, ecard (Set.filter (λx. meval_trm f x = y) group))) ‘ meval_trm f ‘ group
    in k[y:=Some (eval_agg_op ω M)]) ‘ (drop b) ‘ rel)

```

definition (in *maux*) *update_since* :: *args* ⇒ *event_data table* ⇒ *event_data table* ⇒ *ts* ⇒ *'msaux* ⇒ *event_data table* × *'msaux* **where**
update_since *args* *rel1* *rel2* *nt* *aux* =
 (let *aux0* = *join_msaux* *args* *rel1* (*add_new_ts_msaux* *args* *nt* *aux*);
 aux' = *add_new_table_msaux* *args* *rel2* *aux0*
 in (*result_msaux* *args* *aux'*, *aux'*))

definition *lookup* = *Mapping.lookup_default* *empty_table*

fun *ε_lax* **where**

```

ε_lax guard φs (MSkip n) = (if n = 0 then guard else empty_table)
| ε_lax guard φs (MTestPos i) = join guard True (φs ! i)
| ε_lax guard φs (MTestNeg i) = join guard False (φs ! i)
| ε_lax guard φs (MPlus r s) = ε_lax guard φs r ∪ ε_lax guard φs s
| ε_lax guard φs (MTimes r s) = join (ε_lax guard φs r) True (ε_lax guard φs s)
| ε_lax guard φs (MStar r) = guard

```

fun *rε_strict* **where**

```

rε_strict n φs (MSkip m) = (if m = 0 then unit_table n else empty_table)
| rε_strict n φs (MTestPos i) = φs ! i
| rε_strict n φs (MTestNeg i) = (if φs ! i = empty_table then unit_table n else empty_table)
| rε_strict n φs (MPlus r s) = rε_strict n φs r ∪ rε_strict n φs s
| rε_strict n φs (MTimes r s) = ε_lax (rε_strict n φs r) φs s
| rε_strict n φs (MStar r) = unit_table n

```

fun *lε_strict* **where**

```

lε_strict n φs (MSkip m) = (if m = 0 then unit_table n else empty_table)
| lε_strict n φs (MTestPos i) = φs ! i
| lε_strict n φs (MTestNeg i) = (if φs ! i = empty_table then unit_table n else empty_table)
| lε_strict n φs (MPlus r s) = lε_strict n φs r ∪ lε_strict n φs s
| lε_strict n φs (MTimes r s) = ε_lax (lε_strict n φs s) φs r
| lε_strict n φs (MStar r) = unit_table n

```

fun *rδ* :: (*mregex* ⇒ *mregex*) ⇒ (*mregex*, 'a table) *mapping* ⇒ 'a table list ⇒ *mregex* ⇒ 'a table **where**

```

rδ κ X φs (MSkip n) = (case n of 0 ⇒ empty_table | Suc m ⇒ lookup X (κ (MSkip m)))
| rδ κ X φs (MTestPos i) = empty_table
| rδ κ X φs (MTestNeg i) = empty_table
| rδ κ X φs (MPlus r s) = rδ κ X φs r ∪ rδ κ X φs s
| rδ κ X φs (MTimes r s) = rδ (λt. κ (MTimes r t)) X φs s ∪ ε_lax (rδ κ X φs r) φs s
| rδ κ X φs (MStar r) = rδ (λt. κ (MTimes (MStar r) t)) X φs r

```

fun *lδ* :: (*mregex* ⇒ *mregex*) ⇒ (*mregex*, 'a table) *mapping* ⇒ 'a table list ⇒ *mregex* ⇒ 'a table **where**

```

lδ κ X φs (MSkip n) = (case n of 0 ⇒ empty_table | Suc m ⇒ lookup X (κ (MSkip m)))
| lδ κ X φs (MTestPos i) = empty_table
| lδ κ X φs (MTestNeg i) = empty_table
| lδ κ X φs (MPlus r s) = lδ κ X φs r ∪ lδ κ X φs s
| lδ κ X φs (MTimes r s) = lδ (λt. κ (MTimes t s)) X φs r ∪ ε_lax (lδ κ X φs s) φs r
| lδ κ X φs (MStar r) = lδ (λt. κ (MTimes t (MStar r))) X φs r

```

lift_definition *mrtabulate* :: *mregex* list ⇒ (*mregex* ⇒ 'b table) ⇒ (*mregex*, 'b table) *mapping*

is λ*k* *f*. (*map_of* (*List.map_filter* (λ*k*. let *fk* = *f* *k* in if *fk* = *empty_table* then *None* else *Some* (*k*,

$fk)) ks)) \langle \text{proof} \rangle$

lemma *lookup_tabulate*:

distinct xs \implies *lookup* (*mrtabulate xs f*) $x =$ (*if* $x \in \text{set } xs$ *then* $f x$ *else* *empty_table*)

$\langle \text{proof} \rangle$

definition *update_matchP* :: $\text{nat} \Rightarrow \mathcal{I} \Rightarrow \text{mregex} \Rightarrow \text{mregex list} \Rightarrow \text{event_data table list} \Rightarrow \text{ts} \Rightarrow$

event_data mr $\delta\text{aux} \Rightarrow \text{event_data table} \times \text{event_data mr}\delta\text{aux}$ **where**

update_matchP n I mr mrs rels nt aux =

(*let aux* = (*case* [(*t*, *mrtabulate mrs* ($\lambda mr.$

$r\delta \text{ id rel rels } mr \cup (\text{if } t = nt \text{ then } r\varepsilon_strict \ n \ rels \ mr \ \text{else } \{\})$)).

(*t*, *rel*) \leftarrow *aux*, *enat* ($nt - t$) \leq *right I*]

of [] \Rightarrow [(*nt*, *mrtabulate mrs* ($r\varepsilon_strict \ n \ rels$))]

| $x \# \text{aux}' \Rightarrow$ (*if* $\text{fst } x = nt$ *then* $x \# \text{aux}'$

else (*nt*, *mrtabulate mrs* ($r\varepsilon_strict \ n \ rels$)) $\# x \# \text{aux}'$)

in (*foldr* (\cup) [*lookup rel mr.* (*t*, *rel*) \leftarrow *aux*, *left I* \leq $nt - t$] {}, *aux*))

definition *update_matchF_base* **where**

update_matchF_base n I mr mrs rels nt =

(*let X* = *mrtabulate mrs* ($l\varepsilon_strict \ n \ rels$))

in ((*nt*, *rels*, *if left I = 0 then lookup X mr else empty_table*), *X*)

definition *update_matchF_step* **where**

update_matchF_step I mr mrs nt = ($\lambda(t, \text{rels}', \text{rel}) (\text{aux}', X)$).

(*let Y* = *mrtabulate mrs* ($l\delta \text{ id } X \ \text{rels}'$))

in ((*t*, *rels'*, *if mem* ($nt - t$) *I then rel* \cup *lookup Y mr else rel*) $\# \text{aux}'$, *Y*))

definition *update_matchF* :: $\text{nat} \Rightarrow \mathcal{I} \Rightarrow \text{mregex} \Rightarrow \text{mregex list} \Rightarrow \text{event_data table list} \Rightarrow \text{ts} \Rightarrow$

event_data ml $\delta\text{aux} \Rightarrow \text{event_data ml}\delta\text{aux}$ **where**

update_matchF n I mr mrs rels nt aux =

fst (*foldr* (*update_matchF_step I mr mrs nt*) *aux* (*update_matchF_base n I mr mrs rels nt*))

fun *eval_matchF* :: $\mathcal{I} \Rightarrow \text{mregex} \Rightarrow \text{ts} \Rightarrow \text{event_data ml}\delta\text{aux} \Rightarrow \text{event_data table list} \times \text{event_data ml}\delta\text{aux}$ **where**

eval_matchF I mr nt [] = ([], [])

| *eval_matchF I mr nt* ((*t*, *rels*, *rel*) $\# \text{aux}$) = (*if* $t + \text{right } I < nt$ *then*

(*let* (*xs*, *aux*) = *eval_matchF I mr nt aux* *in* (*rel* $\#$ *xs*, *aux*)) *else* ([], (*t*, *rels*, *rel*) $\# \text{aux}$))

primrec *map_split* **where**

map_split f [] = ([], [])

| *map_split f* ($x \# xs$) =

(*let* (*y*, *z*) = $f x$; (*ys*, *zs*) = *map_split f xs*)

in ($y \# ys$, $z \# zs$))

fun *eval_assignment* :: $\text{nat} \times \text{Formula.trm} \Rightarrow \text{event_data tuple} \Rightarrow \text{event_data tuple}$ **where**

eval_assignment (*x*, *t*) *y* = ($y[x := \text{Some} (\text{meval_trm } t \ y)]$)

fun *eval_constraint0* :: $\text{mconstraint} \Rightarrow \text{event_data} \Rightarrow \text{event_data} \Rightarrow \text{bool}$ **where**

eval_constraint0 MEq *x y* = ($x = y$)

| *eval_constraint0 MLess* *x y* = ($x < y$)

| *eval_constraint0 MLessEq* *x y* = ($x \leq y$)

fun *eval_constraint* :: $\text{Formula.trm} \times \text{bool} \times \text{mconstraint} \times \text{Formula.trm} \Rightarrow \text{event_data tuple} \Rightarrow \text{bool}$ **where**

eval_constraint (*t1*, *p*, *c*, *t2*) *x* = (*eval_constraint0 c* (*meval_trm t1 x*) (*meval_trm t2 x*) = *p*)

primrec (**in** *maux*) *meval* :: $\text{nat} \Rightarrow \text{ts} \Rightarrow \text{Formula.database} \Rightarrow ('msaux, 'muaux) \text{mformula} \Rightarrow$

event_data table list $\times ('msaux, 'muaux) \text{mformula}$ **where**

```

    meval n t db (MRel rel) = ([rel], MRel rel)
| meval n t db (MPred e ts) = (map (λX. (λf. Table.tabulate f 0 n) ‘ Option.these
    (match ts ‘ X)) (case Mapping.lookup db e of None ⇒ [[]] | Some xs ⇒ xs), MPred e ts)
| meval n t db (MLet p m φ ψ) =
    (let (xs, φ) = meval m t db φ; (ys, ψ) = meval n t (Mapping.update p (map (image (map the)) xs)
db) ψ
    in (ys, MLet p m φ ψ))
| meval n t db (MAnd A_φ φ pos A_ψ ψ buf) =
    (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
        (zs, buf) = mbuf2_take (λr1 r2. bin_join n A_φ r1 pos A_ψ r2) (mbuf2_add xs ys buf)
    in (zs, MAnd A_φ φ pos A_ψ ψ buf))
| meval n t db (MAndAssign φ conf) =
    (let (xs, φ) = meval n t db φ in (map (λr. eval_assignment conf ‘ r) xs, MAndAssign φ conf))
| meval n t db (MAndRel φ conf) =
    (let (xs, φ) = meval n t db φ in (map (Set.filter (eval_constraint conf)) xs, MAndRel φ conf))
| meval n t db (MAnds A_pos A_neg L buf) =
    (let R = map (meval n t db) L in
        let buf = mbufn_add (map fst R) buf in
        let (zs, buf) = mbufn_take (λxs zs. zs @ [mmulti_join n A_pos A_neg xs]) [] buf in
        (zs, MAnds A_pos A_neg (map snd R) 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 (MNeg φ) =
    (let (xs, φ) = meval n t db φ in (map (λr. (if r = empty_table then unit_table n else empty_table))
xs, MNeg φ))
| 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 (MAgg g0 y ω b f φ) =
    (let (xs, φ) = meval (b + n) t db φ in (map (eval_agg n g0 y ω b f) xs, MAgg g0 y ω b f 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 args φ 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 args r1 r2 t aux
                in (zs @ [z], aux)) ([], aux) (mbuf2_add xs ys buf) (nts @ [t])
    in (zs, MSince args φ ψ buf nts aux))
| meval n t db (MUntil args φ ψ buf nts aux) =
    (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
        (aux, buf, nts) = mbuf2t_take (add_new_muaux args) aux (mbuf2_add xs ys buf) (nts @ [t]);
        (zs, aux) = eval_muaux args (case nts of [] ⇒ t | nt # _ ⇒ nt) aux
    in (zs, MUntil args φ ψ buf nts aux))
| meval n t db (MMatchP I mr mrs φs buf nts aux) =
    (let (xss, φs) = map_split id (map (meval n t db) φs);
        ((zs, aux), buf, nts) = mbufnt_take (λrels t (zs, aux).
            let (z, aux) = update_matchP n I mr mrs rels t aux
                in (zs @ [z], aux)) ([], aux) (mbufn_add xss buf) (nts @ [t])
    in (zs, MMatchP I mr mrs φs buf nts aux))
| meval n t db (MMatchF I mr mrs φs buf nts aux) =

```

```

    (let (xss,  $\varphi$ s) = map_split id (map (meval n t db)  $\varphi$ s);
        (aux, buf, nts) = mbufnt_take (update_matchF n I mr mrs) aux (mbufn_add xss buf) (nts @ [t]);
        (zs, aux) = eval_matchF I mr (case nts of []  $\Rightarrow$  t | nt # _  $\Rightarrow$  nt) aux
    in (zs, MMatchF I mr mrs  $\varphi$ s buf nts aux))

```

definition (in mau x) mstep :: Formula.database \times ts \Rightarrow ('msaux, 'muau x) mstate \Rightarrow (nat \times event_data table) list \times ('msaux, 'muau x) mstate **where**

```

mstep tdb st =
  (let (xs, m) = meval (mstate_n st) (snd tdb) (fst tdb) (mstate_m st)
      in (indexed_from (mstate_i st) xs,
          (mstate_i = mstate_i st + length xs, mstate_m = m, mstate_n = mstate_n st)))

```

6.4 Verdict delay

context fixes σ :: Formula.trace **begin**

```

fun progress :: (Formula.name  $\rightarrow$  nat)  $\Rightarrow$  Formula.formula  $\Rightarrow$  nat  $\Rightarrow$  nat where
  progress P (Formula.Pred e ts) j = (case P e of None  $\Rightarrow$  j | Some k  $\Rightarrow$  k)
| progress P (Formula.Let p  $\varphi$   $\psi$ ) j = progress (P(p  $\mapsto$  progress P  $\varphi$  j))  $\psi$  j
| progress P (Formula.Eq t1 t2) j = j
| progress P (Formula.Less t1 t2) j = j
| progress P (Formula.LessEq t1 t2) j = j
| progress P (Formula.Neg  $\varphi$ ) j = progress P  $\varphi$  j
| progress P (Formula.Or  $\varphi$   $\psi$ ) j = min (progress P  $\varphi$  j) (progress P  $\psi$  j)
| progress P (Formula.And  $\varphi$   $\psi$ ) j = min (progress P  $\varphi$  j) (progress P  $\psi$  j)
| progress P (Formula.Ands l) j = (if l = [] then j else Min (set (map ( $\lambda\varphi$ . progress P  $\varphi$  j) l)))
| progress P (Formula.Exists  $\varphi$ ) j = progress P  $\varphi$  j
| progress P (Formula.Agg y  $\omega$  b f  $\varphi$ ) j = progress P  $\varphi$  j
| progress P (Formula.Prev I  $\varphi$ ) j = (if j = 0 then 0 else min (Suc (progress P  $\varphi$  j)) j)
| progress P (Formula.Next I  $\varphi$ ) j = progress P  $\varphi$  j - 1
| progress P (Formula.Since  $\varphi$  I  $\psi$ ) j = min (progress P  $\varphi$  j) (progress P  $\psi$  j)
| progress P (Formula.Until  $\varphi$  I  $\psi$ ) j =
  Inf {i.  $\forall k$ . k < j  $\wedge$  k  $\leq$  min (progress P  $\varphi$  j) (progress P  $\psi$  j)  $\rightarrow$   $\tau$   $\sigma$  i + right I  $\geq$   $\tau$   $\sigma$  k}
| progress P (Formula.MatchP I r) j = min_regex_default (progress P) r j
| progress P (Formula.MatchF I r) j =
  Inf {i.  $\forall k$ . k < j  $\wedge$  k  $\leq$  min_regex_default (progress P) r j  $\rightarrow$   $\tau$   $\sigma$  i + right I  $\geq$   $\tau$   $\sigma$  k}

```

definition progress_regex P = min_regex_default (progress P)

declare progress.simps[simp del]

lemmas progress_simps[simp] = progress.simps[folded progress_regex_def[THEN fun_cong, THEN fun_cong]]

end

definition pred_mapping Q = pred_fun (λ _. True) (pred_option Q)

definition rel_mapping Q = rel_fun (=) (rel_option Q)

lemma pred_mapping_alt: pred_mapping Q P = ($\forall p \in$ dom P. Q (the (P p)))
(proof)

lemma rel_mapping_alt: rel_mapping Q P P' = (dom P = dom P' \wedge ($\forall p \in$ dom P. Q (the (P p)) (the (P' p))))
(proof)

lemma rel_mapping_map_upd[simp]: Q x y \Longrightarrow rel_mapping Q P P' \Longrightarrow rel_mapping Q (P(p \mapsto x)) (P'(p \mapsto y))
(proof)

lemma *pred_mapping_map_upd[simp]*: $Q\ x \implies \text{pred_mapping}\ Q\ P \implies \text{pred_mapping}\ Q\ (P(p \mapsto x))$
 ⟨proof⟩

lemma *pred_mapping_empty[simp]*: $\text{pred_mapping}\ Q\ \text{Map.empty}$
 ⟨proof⟩

lemma *pred_mapping_mono*: $\text{pred_mapping}\ Q\ P \implies Q \leq R \implies \text{pred_mapping}\ R\ P$
 ⟨proof⟩

lemma *pred_mapping_mono_strong*: $\text{pred_mapping}\ Q\ P \implies$
 $(\bigwedge p. p \in \text{dom}\ P \implies Q\ (\text{the}\ (P\ p)) \implies R\ (\text{the}\ (P\ p))) \implies \text{pred_mapping}\ R\ P$
 ⟨proof⟩

lemma *progress_mono_gen*: $j \leq j' \implies \text{rel_mapping}\ (\leq)\ P\ P' \implies \text{progress}\ \sigma\ P\ \varphi\ j \leq \text{progress}\ \sigma\ P'\ \varphi\ j'$
 ⟨proof⟩

lemma *rel_mapping_reflP*: $\text{reflP}\ Q \implies \text{rel_mapping}\ Q\ P\ P$
 ⟨proof⟩

lemmas *progress_mono = progress_mono_gen*[OF_ *rel_mapping_reflP*[unfolded *reflP_def*], *simplified*]

lemma *progress_le_gen*: $\text{pred_mapping}\ (\lambda x. x \leq j)\ P \implies \text{progress}\ \sigma\ P\ \varphi\ j \leq j$
 ⟨proof⟩

lemma *progress_le*: $\text{progress}\ \sigma\ \text{Map.empty}\ \varphi\ j \leq j$
 ⟨proof⟩

lemma *progress_0_gen[simp]*:
 $\text{pred_mapping}\ (\lambda x. x = 0)\ P \implies \text{progress}\ \sigma\ P\ \varphi\ 0 = 0$
 ⟨proof⟩

lemma *progress_0[simp]*:
 $\text{progress}\ \sigma\ \text{Map.empty}\ \varphi\ 0 = 0$
 ⟨proof⟩

definition *max_mapping* :: $('b \Rightarrow 'a\ \text{option}) \Rightarrow ('b \Rightarrow 'a\ \text{option}) \Rightarrow 'b \Rightarrow ('a :: \text{linorder})\ \text{option}$ **where**
 $\text{max_mapping}\ P\ P'\ x = (\text{case}\ (P\ x, P'\ x)\ \text{of}$
 $(\text{None}, \text{None}) \Rightarrow \text{None}$
 $| (\text{Some}\ x, \text{None}) \Rightarrow \text{Some}\ x$
 $| (\text{None}, \text{Some}\ x) \Rightarrow \text{Some}\ x$
 $| (\text{Some}\ x, \text{Some}\ y) \Rightarrow \text{Some}\ (\text{max}\ x\ y))$

definition *Max_mapping* :: $('b \Rightarrow 'a\ \text{option})\ \text{set} \Rightarrow 'b \Rightarrow ('a :: \text{linorder})\ \text{option}$ **where**
 $\text{Max_mapping}\ Ps\ x = (\text{if}\ (\forall P \in Ps. P\ x \neq \text{None})\ \text{then}\ \text{Some}\ (\text{Max}\ ((\lambda P. \text{the}\ (P\ x))\ 'Ps))\ \text{else}\ \text{None})$

lemma *dom_max_mapping[simp]*: $\text{dom}\ (\text{max_mapping}\ P1\ P2) = \text{dom}\ P1 \cap \text{dom}\ P2$
 ⟨proof⟩

lemma *dom_Max_mapping[simp]*: $\text{dom}\ (\text{Max_mapping}\ X) = (\bigcap P \in X. \text{dom}\ P)$
 ⟨proof⟩

lemma *Max_mapping_coboundedI*:
assumes *finite* $X \forall Q \in X. \text{dom}\ Q = \text{dom}\ P\ P \in X$
shows $\text{rel_mapping}\ (\leq)\ P\ (\text{Max_mapping}\ X)$
 ⟨proof⟩

lemma *rel_mapping_trans*: $P\ \text{OO}\ Q \leq R \implies$

$rel_mapping\ P\ P1\ P2 \implies rel_mapping\ Q\ P2\ P3 \implies rel_mapping\ R\ P1\ P3$
 ⟨proof⟩

abbreviation $range_mapping :: nat \Rightarrow nat \Rightarrow ('b \Rightarrow nat\ option) \Rightarrow bool$ **where**
 $range_mapping\ i\ j\ P \equiv pred_mapping\ (\lambda x. i \leq x \wedge x \leq j)\ P$

lemma $range_mapping_relax$:
 $range_mapping\ i\ j\ P \implies i' \leq i \implies j' \geq j \implies range_mapping\ i'\ j'\ P$
 ⟨proof⟩

lemma $range_mapping_max_mapping[simp]$:
 $range_mapping\ i\ j1\ P1 \implies range_mapping\ i\ j2\ P2 \implies range_mapping\ i\ (max\ j1\ j2)\ (max_mapping\ P1\ P2)$
 ⟨proof⟩

lemma $range_mapping_Max_mapping[simp]$:
 $finite\ X \implies X \neq \{\}\ \implies \forall x \in X. range_mapping\ i\ (j\ x)\ (P\ x) \implies range_mapping\ i\ (Max\ (j\ 'X))\ (Max_mapping\ (P\ 'X))$
 ⟨proof⟩

lemma $pred_mapping_le$:
 $pred_mapping\ ((\leq)\ i)\ P1 \implies rel_mapping\ (\leq)\ P1\ P2 \implies pred_mapping\ ((\leq)\ (i :: nat))\ P2$
 ⟨proof⟩

lemma $pred_mapping_le'$:
 $pred_mapping\ ((\leq)\ j)\ P1 \implies i \leq j \implies rel_mapping\ (\leq)\ P1\ P2 \implies pred_mapping\ ((\leq)\ (i :: nat))\ P2$
 ⟨proof⟩

lemma $max_mapping_cobounded1$: $dom\ P1 \subseteq dom\ P2 \implies rel_mapping\ (\leq)\ P1\ (max_mapping\ P1\ P2)$
 ⟨proof⟩

lemma $max_mapping_cobounded2$: $dom\ P2 \subseteq dom\ P1 \implies rel_mapping\ (\leq)\ P2\ (max_mapping\ P1\ P2)$
 ⟨proof⟩

lemma $max_mapping_fun_upd2[simp]$:
 $(max_mapping\ P1\ (P2(p := y)))(p \mapsto x) = (max_mapping\ P1\ P2)(p \mapsto x)$
 ⟨proof⟩

lemma $rel_mapping_max_mapping_fun_upd$: $dom\ P2 \subseteq dom\ P1 \implies p \in dom\ P2 \implies the\ (P2\ p) \leq y \implies rel_mapping\ (\leq)\ P2\ ((max_mapping\ P1\ P2)(p \mapsto y))$
 ⟨proof⟩

lemma $progress_ge_gen$: $Formula.future_bounded\ \varphi \implies \exists P\ j. dom\ P = S \wedge range_mapping\ i\ j\ P \wedge i \leq progress\ \sigma\ P\ \varphi\ j$
 ⟨proof⟩

lemma $progress_ge$: $Formula.future_bounded\ \varphi \implies \exists j. i \leq progress\ \sigma\ Map.empty\ \varphi\ j$
 ⟨proof⟩

lemma $cInf_restrict_nat$:
fixes $x :: nat$
assumes $x \in A$
shows $Inf\ A = Inf\ \{y \in A. y \leq x\}$
 ⟨proof⟩

lemma *progress_time_conv*:

assumes $\forall i < j. \tau \sigma i = \tau \sigma' i$
shows $\text{progress } \sigma P \varphi j = \text{progress } \sigma' P \varphi j$
(*proof*)

lemma *Inf_UNIV_nat*: $(\text{Inf UNIV} :: \text{nat}) = 0$

(*proof*)

lemma *progress_prefix_conv*:

assumes *prefix_of* $\pi \sigma$ **and** *prefix_of* $\pi \sigma'$
shows $\text{progress } \sigma P \varphi (\text{plen } \pi) = \text{progress } \sigma' P \varphi (\text{plen } \pi)$
(*proof*)

lemma *bounded_rtranclp_mono*:

fixes $n :: 'x :: \text{linorder}$
assumes $\bigwedge i j. R i j \implies j < n \implies S i j \wedge i j. R i j \implies i \leq j$
shows $\text{rtranclp } R i j \implies j < n \implies \text{rtranclp } S i j$
(*proof*)

lemma *sat_prefix_conv_gen*:

assumes *prefix_of* $\pi \sigma$ **and** *prefix_of* $\pi \sigma'$
shows $i < \text{progress } \sigma P \varphi (\text{plen } \pi) \implies \text{dom } V = \text{dom } V' \implies \text{dom } P = \text{dom } V \implies$
 $\text{pred_mapping } (\lambda x. x \leq \text{plen } \pi) P \implies$
 $(\bigwedge p i \varphi. p \in \text{dom } V \implies i < \text{the } (P p) \implies \text{the } (V p) i = \text{the } (V' p) i) \implies$
 $\text{Formula.sat } \sigma V v i \varphi \longleftrightarrow \text{Formula.sat } \sigma' V' v i \varphi$
(*proof*)

lemma *sat_prefix_conv*:

assumes *prefix_of* $\pi \sigma$ **and** *prefix_of* $\pi \sigma'$
shows $i < \text{progress } \sigma \text{Map.empty } \varphi (\text{plen } \pi) \implies$
 $\text{Formula.sat } \sigma \text{Map.empty } v i \varphi \longleftrightarrow \text{Formula.sat } \sigma' \text{Map.empty } v i \varphi$
(*proof*)

lemma *progress_remove_neg[simp]*: $\text{progress } \sigma P (\text{remove_neg } \varphi) j = \text{progress } \sigma P \varphi j$

(*proof*)

lemma *safe_progress_get_and*: $\text{safe_formula } \varphi \implies$

$\text{Min } ((\lambda \varphi. \text{progress } \sigma P \varphi j) \text{ `set } (\text{get_and_list } \varphi)) = \text{progress } \sigma P \varphi j$
(*proof*)

lemma *progress_convert_multiway*: $\text{safe_formula } \varphi \implies \text{progress } \sigma P (\text{convert_multiway } \varphi) j = \text{progress } \sigma P \varphi j$

(*proof*)

6.5 Specification

definition *pprogress* :: $\text{Formula.formula} \Rightarrow \text{Formula.prefix} \Rightarrow \text{nat}$ **where**

$\text{pprogress } \varphi \pi = (\text{THE } n. \forall \sigma. \text{prefix_of } \pi \sigma \longrightarrow \text{progress } \sigma \text{Map.empty } \varphi (\text{plen } \pi) = n)$

lemma *pprogress_eq*: $\text{prefix_of } \pi \sigma \implies \text{pprogress } \varphi \pi = \text{progress } \sigma \text{Map.empty } \varphi (\text{plen } \pi)$

(*proof*)

locale *future_bounded_mfodl* =

fixes $\varphi :: \text{Formula.formula}$
assumes *future_bounded*: $\text{Formula.future_bounded } \varphi$

sublocale *future_bounded_mfodl* \subseteq *sliceable_timed_progress* $\text{Formula.nfv } \varphi \text{Formula.fv } \varphi \text{relevant_events}$

φ
 $\lambda \sigma v i. \text{Formula.sat } \sigma \text{ Map.empty } v i \varphi \text{ pprogress } \varphi$
 $\langle \text{proof} \rangle$

locale *verimon_spec* =
fixes $\varphi :: \text{Formula.formula}$
assumes *monitorable*: *mmonitorable* φ

sublocale *verimon_spec* \subseteq *future_bounded_mfodl*
 $\langle \text{proof} \rangle$

6.6 Correctness

6.6.1 Invariants

definition *wf_mbuf2* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{event_data table} \Rightarrow \text{bool}) \Rightarrow (\text{nat} \Rightarrow \text{event_data table} \Rightarrow \text{bool}) \Rightarrow$
 $\text{event_data mbuf2} \Rightarrow \text{bool}$ **where**
 $\text{wf_mbuf2 } i \text{ ja } \text{jb } P \ Q \ \text{buf} \longleftrightarrow i \leq \text{ja} \wedge i \leq \text{jb} \wedge (\text{case } \text{buf} \text{ of } (xs, ys) \Rightarrow$
 $\text{list_all2 } P \ [i..<\text{ja}] \ xs \wedge \text{list_all2 } Q \ [i..<\text{jb}] \ ys)$

inductive *list_all3* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'c \text{ list} \Rightarrow \text{bool}$ **for** $P :: ('a \Rightarrow 'b \Rightarrow 'c \Rightarrow \text{bool})$ **where**
 $\text{list_all3 } P \ [] \ [] \ []$
 $| P \ a1 \ a2 \ a3 \Longrightarrow \text{list_all3 } P \ q1 \ q2 \ q3 \Longrightarrow \text{list_all3 } P \ (a1 \ \# \ q1) \ (a2 \ \# \ q2) \ (a3 \ \# \ q3)$

lemma *list_all3_list_all2D*: $\text{list_all3 } P \ xs \ ys \ zs \Longrightarrow$
 $(\text{length } xs = \text{length } ys \wedge \text{list_all2 } (\text{case_prod } P) \ (\text{zip } xs \ ys) \ zs)$
 $\langle \text{proof} \rangle$

lemma *list_all2_list_all3I*: $\text{length } xs = \text{length } ys \Longrightarrow \text{list_all2 } (\text{case_prod } P) \ (\text{zip } xs \ ys) \ zs \Longrightarrow$
 $\text{list_all3 } P \ xs \ ys \ zs$
 $\langle \text{proof} \rangle$

lemma *list_all3_list_all2_eq*: $\text{list_all3 } P \ xs \ ys \ zs \longleftrightarrow$
 $(\text{length } xs = \text{length } ys \wedge \text{list_all2 } (\text{case_prod } P) \ (\text{zip } xs \ ys) \ zs)$
 $\langle \text{proof} \rangle$

lemma *list_all3_mapD*: $\text{list_all3 } P \ (\text{map } f \ xs) \ (\text{map } g \ ys) \ (\text{map } h \ zs) \Longrightarrow$
 $\text{list_all3 } (\lambda x \ y \ z. P \ (f \ x) \ (g \ y) \ (h \ z)) \ xs \ ys \ zs$
 $\langle \text{proof} \rangle$

lemma *list_all3_mapI*: $\text{list_all3 } (\lambda x \ y \ z. P \ (f \ x) \ (g \ y) \ (h \ z)) \ xs \ ys \ zs \Longrightarrow$
 $\text{list_all3 } P \ (\text{map } f \ xs) \ (\text{map } g \ ys) \ (\text{map } h \ zs)$
 $\langle \text{proof} \rangle$

lemma *list_all3_map_iff*: $\text{list_all3 } P \ (\text{map } f \ xs) \ (\text{map } g \ ys) \ (\text{map } h \ zs) \longleftrightarrow$
 $\text{list_all3 } (\lambda x \ y \ z. P \ (f \ x) \ (g \ y) \ (h \ z)) \ xs \ ys \ zs$
 $\langle \text{proof} \rangle$

lemmas *list_all3_map* =
 $\text{list_all3_map_iff}[\text{where } g=id \ \text{and } h=id, \text{unfolded } \text{list.map_id } id_apply]$
 $\text{list_all3_map_iff}[\text{where } f=id \ \text{and } h=id, \text{unfolded } \text{list.map_id } id_apply]$
 $\text{list_all3_map_iff}[\text{where } f=id \ \text{and } g=id, \text{unfolded } \text{list.map_id } id_apply]$

lemma *list_all3_conv_all_nth*:
 $\text{list_all3 } P \ xs \ ys \ zs =$
 $(\text{length } xs = \text{length } ys \wedge \text{length } ys = \text{length } zs \wedge (\forall i < \text{length } xs. P \ (xs!i) \ (ys!i) \ (zs!i)))$
 $\langle \text{proof} \rangle$

lemma *list_all3_refl* [intro?]:
 $(\bigwedge x. x \in \text{set } xs \implies P x x x) \implies \text{list_all3 } P \text{ } xs \text{ } xs \text{ } xs$
 ⟨proof⟩

definition *wf_mbufn* :: $\text{nat} \Rightarrow \text{nat list} \Rightarrow (\text{nat} \Rightarrow \text{event_data table} \Rightarrow \text{bool}) \text{ list} \Rightarrow \text{event_data mbufn} \Rightarrow \text{bool}$ **where**
 $\text{wf_mbufn } i \text{ } js \text{ } Ps \text{ } buf \iff \text{list_all3 } (\lambda P j \text{ } xs. i \leq j \wedge \text{list_all2 } P [i..<j] \text{ } xs) \text{ } Ps \text{ } js \text{ } buf$

definition *wf_mbuf2'* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{event_data list set} \Rightarrow \text{Formula.formula} \Rightarrow \text{Formula.formula} \Rightarrow \text{event_data mbuf2} \Rightarrow \text{bool}$ **where**
 $\text{wf_mbuf2}' \sigma \text{ } P \text{ } V \text{ } j \text{ } n \text{ } R \text{ } \varphi \text{ } \psi \text{ } buf \iff \text{wf_mbuf2 } (\text{min } (\text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j) \text{ } (\text{progress } \sigma \text{ } P \text{ } \psi \text{ } j))$
 $(\text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j) \text{ } (\text{progress } \sigma \text{ } P \text{ } \psi \text{ } j)$
 $(\lambda i. \text{qtable } n \text{ } (\text{Formula.fv } \varphi) \text{ } (\text{mem_restr } R) \text{ } (\lambda v. \text{Formula.sat } \sigma \text{ } V \text{ } (\text{map the } v) \text{ } i \text{ } \varphi))$
 $(\lambda i. \text{qtable } n \text{ } (\text{Formula.fv } \psi) \text{ } (\text{mem_restr } R) \text{ } (\lambda v. \text{Formula.sat } \sigma \text{ } V \text{ } (\text{map the } v) \text{ } i \text{ } \psi)) \text{ } buf$

definition *wf_mbufn'* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{event_data list set} \Rightarrow \text{Formula.formula} \text{ } \text{Regex.regex} \Rightarrow \text{event_data mbufn} \Rightarrow \text{bool}$ **where**
 $\text{wf_mbufn}' \sigma \text{ } P \text{ } V \text{ } j \text{ } n \text{ } R \text{ } r \text{ } buf \iff (\text{case to_mregex } r \text{ } \text{of } (mr, \varphi s) \Rightarrow$
 $\text{wf_mbufn } (\text{progress_regex } \sigma \text{ } P \text{ } r \text{ } j) \text{ } (\text{map } (\lambda \varphi. \text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j) \text{ } \varphi s)$
 $(\text{map } (\lambda \varphi \text{ } i. \text{qtable } n \text{ } (\text{Formula.fv } \varphi) \text{ } (\text{mem_restr } R) \text{ } (\lambda v. \text{Formula.sat } \sigma \text{ } V \text{ } (\text{map the } v) \text{ } i \text{ } \varphi)) \text{ } \varphi s)$
 $buf)$

lemma *wf_mbuf2'_UNIV_alt*: $\text{wf_mbuf2}' \sigma \text{ } P \text{ } V \text{ } j \text{ } n \text{ } UNIV \text{ } \varphi \text{ } \psi \text{ } buf \iff (\text{case } buf \text{ } \text{of } (xs, ys) \Rightarrow$
 $\text{list_all2 } (\lambda i. \text{wf_table } n \text{ } (\text{Formula.fv } \varphi) \text{ } (\lambda v. \text{Formula.sat } \sigma \text{ } V \text{ } (\text{map the } v) \text{ } i \text{ } \varphi))$
 $[\text{min } (\text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j) \text{ } (\text{progress } \sigma \text{ } P \text{ } \psi \text{ } j) \text{ } ..< \text{ } (\text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j)] \text{ } xs \wedge$
 $\text{list_all2 } (\lambda i. \text{wf_table } n \text{ } (\text{Formula.fv } \psi) \text{ } (\lambda v. \text{Formula.sat } \sigma \text{ } V \text{ } (\text{map the } v) \text{ } i \text{ } \psi))$
 $[\text{min } (\text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j) \text{ } (\text{progress } \sigma \text{ } P \text{ } \psi \text{ } j) \text{ } ..< \text{ } (\text{progress } \sigma \text{ } P \text{ } \psi \text{ } j)] \text{ } ys)$
 ⟨proof⟩

definition *wf_ts* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{Formula.formula} \Rightarrow \text{Formula.formula} \Rightarrow \text{ts list} \Rightarrow \text{bool}$ **where**
 $\text{wf_ts } \sigma \text{ } P \text{ } j \text{ } \varphi \text{ } \psi \text{ } ts \iff \text{list_all2 } (\lambda i \text{ } t. t = \tau \sigma \text{ } i) [\text{min } (\text{progress } \sigma \text{ } P \text{ } \varphi \text{ } j) \text{ } (\text{progress } \sigma \text{ } P \text{ } \psi \text{ } j) \text{ } ..< j] \text{ } ts$

definition *wf_ts_regex* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{Formula.formula} \text{ } \text{Regex.regex} \Rightarrow \text{ts list} \Rightarrow \text{bool}$ **where**
 $\text{wf_ts_regex } \sigma \text{ } P \text{ } j \text{ } r \text{ } ts \iff \text{list_all2 } (\lambda i \text{ } t. t = \tau \sigma \text{ } i) [\text{progress_regex } \sigma \text{ } P \text{ } r \text{ } j \text{ } ..< j] \text{ } ts$

abbreviation *Since_pos* $\varphi \text{ } I \text{ } \psi \equiv \text{Formula.Since } (\text{if } pos \text{ then } \varphi \text{ else } \text{Formula.Neg } \varphi) \text{ } I \text{ } \psi$

definition (in *msaux*) *wf_since_aux* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{event_data list set} \Rightarrow \text{args} \Rightarrow \text{Formula.formula} \Rightarrow \text{Formula.formula} \Rightarrow \text{'msaux} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{wf_since_aux } \sigma \text{ } V \text{ } R \text{ } \text{args } \varphi \text{ } \psi \text{ } aux \text{ } ne \iff \text{Formula.fv } \varphi \subseteq \text{Formula.fv } \psi \wedge (\exists \text{cur } auxlist. \text{valid_msaux } \text{args } \text{cur } aux \text{ } auxlist \wedge$
 $\text{cur} = (\text{if } ne = 0 \text{ then } 0 \text{ else } \tau \sigma \text{ } (ne - 1)) \wedge$
 $\text{sorted_wrt } (\lambda x \text{ } y. \text{fst } x > \text{fst } y) \text{ } auxlist \wedge$
 $(\forall t \text{ } X. (t, X) \in \text{set } auxlist \longrightarrow ne \neq 0 \wedge t \leq \tau \sigma \text{ } (ne - 1) \wedge \tau \sigma \text{ } (ne - 1) - t \leq \text{right } (\text{args_ivl } \text{args}) \wedge (\exists i. \tau \sigma \text{ } i = t) \wedge$
 $\text{qtable } (\text{args_n } \text{args}) \text{ } (\text{Formula.fv } \psi) \text{ } (\text{mem_restr } R) \text{ } (\lambda v. \text{Formula.sat } \sigma \text{ } V \text{ } (\text{map the } v) \text{ } (ne - 1)$
 $(\text{Since_pos } (\text{args_pos } \text{args}) \text{ } \varphi \text{ } (\text{point } (\tau \sigma \text{ } (ne - 1) - t)) \text{ } \psi)) \text{ } X) \wedge$
 $(\forall t. ne \neq 0 \wedge t \leq \tau \sigma \text{ } (ne - 1) \wedge \tau \sigma \text{ } (ne - 1) - t \leq \text{right } (\text{args_ivl } \text{args}) \wedge (\exists i. \tau \sigma \text{ } i = t) \longrightarrow$
 $(\exists X. (t, X) \in \text{set } auxlist)))$

definition *wf_matchP_aux* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{event_data list set} \Rightarrow \mathcal{I} \Rightarrow \text{Formula.formula} \text{ } \text{Regex.regex} \Rightarrow \text{event_data mrdaux} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{wf_matchP_aux } \sigma \text{ } V \text{ } n \text{ } R \text{ } I \text{ } r \text{ } aux \text{ } ne \iff \text{sorted_wrt } (\lambda x \text{ } y. \text{fst } x > \text{fst } y) \text{ } aux \wedge$
 $(\forall t \text{ } X. (t, X) \in \text{set } aux \longrightarrow ne \neq 0 \wedge t \leq \tau \sigma \text{ } (ne - 1) \wedge \tau \sigma \text{ } (ne - 1) - t \leq \text{right } I \wedge (\exists i. \tau \sigma \text{ } i =$
 $t) \wedge$

$(\text{case to_mregex } r \text{ of } (mr, \varphi s) \Rightarrow$
 $(\forall ms \in \text{RPDs } mr. \text{qtable } n (\text{Formula.fv_regex } r) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v)$
 $(ne-1)$
 $(\text{Formula.MatchP } (\text{point } (\tau \sigma (ne-1) - t)) (\text{from_mregex } ms \varphi s)))$
 $(\text{lookup } X ms))) \wedge$
 $(\forall t. ne \neq 0 \wedge t \leq \tau \sigma (ne-1) \wedge \tau \sigma (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \sigma i = t) \longrightarrow$
 $(\exists X. (t, X) \in \text{set } aux))$

lemma *qtable_mem_restr_UNIV*: $\text{qtable } n A (\text{mem_restr } UNIV) Q X = \text{wf_table } n A Q X$
 $\langle \text{proof} \rangle$

lemma (in *msaux*) *wf_since_aux_UNIV_alt*:

$\text{wf_since_aux } \sigma V UNIV \text{ args } \varphi \psi \text{ aux } ne \longleftrightarrow \text{Formula.fv } \varphi \subseteq \text{Formula.fv } \psi \wedge (\exists \text{cur } \text{auxlist.}$
 $\text{valid_msaux } \text{args } \text{cur } \text{aux } \text{auxlist } \wedge$
 $\text{cur} = (\text{if } ne = 0 \text{ then } 0 \text{ else } \tau \sigma (ne - 1)) \wedge$
 $\text{sorted_wrt } (\lambda x y. \text{fst } x > \text{fst } y) \text{auxlist } \wedge$
 $(\forall t X. (t, X) \in \text{set } \text{auxlist} \longrightarrow ne \neq 0 \wedge t \leq \tau \sigma (ne - 1) \wedge \tau \sigma (ne - 1) - t \leq \text{right } (\text{args_ivl}$
 $\text{args}) \wedge (\exists i. \tau \sigma i = t) \wedge$
 $\text{wf_table } (\text{args_n } \text{args}) (\text{Formula.fv } \psi)$
 $(\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) (ne - 1) (\text{Sincep } (\text{args_pos } \text{args}) \varphi (\text{point } (\tau \sigma (ne - 1) -$
 $t)) \psi)) X) \wedge$
 $(\forall t. ne \neq 0 \wedge t \leq \tau \sigma (ne - 1) \wedge \tau \sigma (ne - 1) - t \leq \text{right } (\text{args_ivl } \text{args}) \wedge (\exists i. \tau \sigma i = t) \longrightarrow$
 $(\exists X. (t, X) \in \text{set } \text{auxlist}))$
 $\langle \text{proof} \rangle$

definition *wf_until_auxlist* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{event_data list set} \Rightarrow \text{bool} \Rightarrow$

$\text{Formula.formula} \Rightarrow \mathcal{I} \Rightarrow \text{Formula.formula} \Rightarrow \text{event_data } \text{muaux} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{wf_until_auxlist } \sigma V n R \text{ pos } \varphi I \psi \text{ auxlist } ne \longleftrightarrow$
 $\text{list_all2 } (\lambda x i. \text{case } x \text{ of } (t, r1, r2) \Rightarrow t = \tau \sigma i \wedge$
 $\text{qtable } n (\text{Formula.fv } \varphi) (\text{mem_restr } R) (\lambda v. \text{if } \text{pos} \text{ then } (\forall k \in \{i..<ne+\text{length } \text{auxlist}\}. \text{Formula.sat}$
 $\sigma V (\text{map the } v) k \varphi)$
 $\text{else } (\exists k \in \{i..<ne+\text{length } \text{auxlist}\}. \text{Formula.sat } \sigma V (\text{map the } v) k \varphi)) r1 \wedge$
 $\text{qtable } n (\text{Formula.fv } \psi) (\text{mem_restr } R) (\lambda v. (\exists j. i \leq j \wedge j < ne + \text{length } \text{auxlist} \wedge \text{mem } (\tau \sigma j -$
 $\tau \sigma i) I \wedge$
 $\text{Formula.sat } \sigma V (\text{map the } v) j \psi \wedge$
 $(\forall k \in \{i..<j\}. \text{if } \text{pos} \text{ then } \text{Formula.sat } \sigma V (\text{map the } v) k \varphi \text{ else } \neg \text{Formula.sat } \sigma V (\text{map the } v)$
 $k \varphi))) r2)$
 $\text{auxlist } [ne..<ne+\text{length } \text{auxlist}]$

definition (in *muaux*) *wf_until_aux* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{event_data list set} \Rightarrow \text{args} \Rightarrow$

$\text{Formula.formula} \Rightarrow \text{Formula.formula} \Rightarrow \text{'muaux} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{wf_until_aux } \sigma V R \text{ args } \varphi \psi \text{ aux } ne \longleftrightarrow \text{Formula.fv } \varphi \subseteq \text{Formula.fv } \psi \wedge$
 $(\exists \text{cur } \text{auxlist. valid_muaux } \text{args } \text{cur } \text{aux } \text{auxlist } \wedge$
 $\text{cur} = (\text{if } ne + \text{length } \text{auxlist} = 0 \text{ then } 0 \text{ else } \tau \sigma (ne + \text{length } \text{auxlist} - 1)) \wedge$
 $\text{wf_until_auxlist } \sigma V (\text{args_n } \text{args}) R (\text{args_pos } \text{args}) \varphi (\text{args_ivl } \text{args}) \psi \text{auxlist } ne)$

lemma (in *muaux*) *wf_until_aux_UNIV_alt*:

$\text{wf_until_aux } \sigma V UNIV \text{ args } \varphi \psi \text{ aux } ne \longleftrightarrow \text{Formula.fv } \varphi \subseteq \text{Formula.fv } \psi \wedge$
 $(\exists \text{cur } \text{auxlist. valid_muaux } \text{args } \text{cur } \text{aux } \text{auxlist } \wedge$
 $\text{cur} = (\text{if } ne + \text{length } \text{auxlist} = 0 \text{ then } 0 \text{ else } \tau \sigma (ne + \text{length } \text{auxlist} - 1)) \wedge$
 $\text{list_all2 } (\lambda x i. \text{case } x \text{ of } (t, r1, r2) \Rightarrow t = \tau \sigma i \wedge$
 $\text{wf_table } (\text{args_n } \text{args}) (\text{Formula.fv } \varphi) (\lambda v. \text{if } (\text{args_pos } \text{args})$
 $\text{then } (\forall k \in \{i..<ne+\text{length } \text{auxlist}\}. \text{Formula.sat } \sigma V (\text{map the } v) k \varphi)$
 $\text{else } (\exists k \in \{i..<ne+\text{length } \text{auxlist}\}. \text{Formula.sat } \sigma V (\text{map the } v) k \varphi)) r1 \wedge$
 $\text{wf_table } (\text{args_n } \text{args}) (\text{Formula.fv } \psi) (\lambda v. \exists j. i \leq j \wedge j < ne + \text{length } \text{auxlist} \wedge \text{mem } (\tau \sigma j - \tau$
 $\sigma i) (\text{args_ivl } \text{args}) \wedge$
 $\text{Formula.sat } \sigma V (\text{map the } v) j \psi \wedge$
 $(\forall k \in \{i..<j\}. \text{if } (\text{args_pos } \text{args}) \text{ then } \text{Formula.sat } \sigma V (\text{map the } v) k \varphi \text{ else } \neg \text{Formula.sat } \sigma V$

(map the v) k φ)) r2)
 auxlist [ne..<ne+length auxlist])
 <proof>

definition wf_matchF_aux :: Formula.trace ⇒ _ ⇒ nat ⇒ event_data list set ⇒
 I ⇒ Formula.formula Regex.regex ⇒ event_data mlδaux ⇒ nat ⇒ nat ⇒ bool **where**
 wf_matchF_aux σ V n R I r aux ne k ←→ (case to_mregex r of (mr, φs) ⇒
 list_all2 (λx i. case x of (t, rels, rel) ⇒ t = τ σ i ∧
 list_all2 (λφ. qtable n (Formula.fv φ) (mem_restr R) (λv.
 Formula.sat σ V (map the v) i φ)) φs rels ∧
 qtable n (Formula.fv_regex r) (mem_restr R) (λv. (∃j. i ≤ j ∧ j < ne + length aux + k ∧ mem
 (τ σ j - τ σ i) I ∧
 Regex.match (Formula.sat σ V (map the v)) r i j)) rel)
 aux [ne..<ne+length aux])

definition wf_matchF_invar **where**
 wf_matchF_invar σ V n R I r st i =
 (case st of (aux, Y) ⇒ aux ≠ [] ∧ wf_matchF_aux σ V n R I r aux i 0 ∧
 (case to_mregex r of (mr, φs) ⇒ ∀ ms ∈ LPDs mr.
 qtable n (Formula.fv_regex r) (mem_restr R) (λv.
 Regex.match (Formula.sat σ V (map the v)) (from_mregex ms φs) i (i + length aux - 1)) (lookup
 Y ms)))

definition lift_envs' :: nat ⇒ event_data list set ⇒ event_data list set **where**
 lift_envs' b R = (λ(xs,ys). xs @ ys) '({xs. length xs = b} × R)

fun formula_of_constraint :: Formula.trm × bool × mconstraint × Formula.trm ⇒ Formula.formula
where
 formula_of_constraint (t1, True, MEq, t2) = Formula.Eq t1 t2
 | formula_of_constraint (t1, True, MLess, t2) = Formula.Less t1 t2
 | formula_of_constraint (t1, True, MLessEq, t2) = Formula.LessEq t1 t2
 | formula_of_constraint (t1, False, MEq, t2) = Formula.Neg (Formula.Eq t1 t2)
 | formula_of_constraint (t1, False, MLess, t2) = Formula.Neg (Formula.Less t1 t2)
 | formula_of_constraint (t1, False, MLessEq, t2) = Formula.Neg (Formula.LessEq t1 t2)

inductive (in maux) wf_mformula :: Formula.trace ⇒ nat ⇒ _ ⇒ _ ⇒
 nat ⇒ event_data list set ⇒ ('msaux, 'muaux) mformula ⇒ Formula.formula ⇒ bool
for σ j **where**
 Eq: is_simple_eq t1 t2 ⇒
 ∀ x ∈ Formula.fv_trm t1. x < n ⇒ ∀ x ∈ Formula.fv_trm t2. x < n ⇒
 wf_mformula σ j P V n R (MRel (eq_rel n t1 t2)) (Formula.Eq t1 t2)
 | neq_Var: x < n ⇒
 wf_mformula σ j P V n R (MRel empty_table) (Formula.Neg (Formula.Eq (Formula.Var x) (Formula.Var
 x)))
 | Pred: ∀ x ∈ Formula.fv (Formula.Pred e ts). x < n ⇒
 ∀ t ∈ set ts. Formula.is_Var t ∨ Formula.is_Const t ⇒
 wf_mformula σ j P V n R (MPred e ts) (Formula.Pred e ts)
 | Let: wf_mformula σ j P V m UNIV φ φ' ⇒
 wf_mformula σ j (P(p ↦ progress σ P φ' j))
 (V(p ↦ λi. {v. length v = m ∧ Formula.sat σ V v i φ'})) n R ψ ψ' ⇒
 {0..<m} ⊆ Formula.fv φ' ⇒ b ≤ m ⇒ m = Formula.nfv φ' ⇒
 wf_mformula σ j P V n R (MLet p m φ ψ) (Formula.Let p φ' ψ')
 | And: wf_mformula σ j P V n R φ φ' ⇒ wf_mformula σ j P V n R ψ ψ' ⇒
 if pos then χ = Formula.And φ' ψ'
 else χ = Formula.And φ' (Formula.Neg ψ') ∧ Formula.fv ψ' ⊆ Formula.fv φ' ⇒
 wf_mbuf2' σ P V j n R φ' ψ' buf ⇒
 wf_mformula σ j P V n R (MAnd (fv φ') φ pos (fv ψ') ψ buf) χ
 | AndAssign: wf_mformula σ j P V n R φ φ' ⇒

$x < n \implies x \notin \text{Formula.fv } \varphi' \implies \text{Formula.fv_trm } t \subseteq \text{Formula.fv } \varphi' \implies (x, t) = \text{conf} \implies$
 $\psi' = \text{Formula.Eq } (\text{Formula.Var } x) t \vee \psi' = \text{Formula.Eq } t (\text{Formula.Var } x) \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MAndAssign } \varphi \text{ conf}) (\text{Formula.And } \varphi' \psi')$
| $\text{AndRel: wf_mformula } \sigma j P V n R \varphi \varphi' \implies$
 $\psi' = \text{formula_of_constraint } \text{conf} \implies$
 $(\text{let } (t1, _, _, t2) = \text{conf} \text{ in } \text{Formula.fv_trm } t1 \cup \text{Formula.fv_trm } t2 \subseteq \text{Formula.fv } \varphi') \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MAndRel } \varphi \text{ conf}) (\text{Formula.And } \varphi' \psi')$
| $\text{AndS: list_all2 } (\lambda \varphi \varphi'. \text{wf_mformula } \sigma j P V n R \varphi \varphi') l (\text{l_pos } @ \text{map } \text{remove_neg } \text{l_neg}) \implies$
 $\text{wf_mbufn } (\text{progress } \sigma P (\text{Formula.Ands } l') j) (\text{map } (\lambda \psi. \text{progress } \sigma P \psi j) (\text{l_pos } @ \text{map } \text{remove_neg } \text{l_neg}))$
 $(\text{map } (\lambda \psi i. \text{qtable } n (\text{Formula.fv } \psi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map } \text{the } v) i \psi)) (\text{l_pos } @ \text{map } \text{remove_neg } \text{l_neg})) \text{ buf} \implies$
 $(\text{l_pos}, \text{l_neg}) = \text{partition safe_formula } l' \implies$
 $\text{l_pos} \neq [] \implies$
 $\text{list_all safe_formula } (\text{map } \text{remove_neg } \text{l_neg}) \implies$
 $A_pos = \text{map } \text{fv } \text{l_pos} \implies$
 $A_neg = \text{map } \text{fv } \text{l_neg} \implies$
 $\bigcup (\text{set } A_neg) \subseteq \bigcup (\text{set } A_pos) \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MAnds } A_pos A_neg l \text{ buf}) (\text{Formula.Ands } l')$
| $\text{Or: wf_mformula } \sigma j P V n R \varphi \varphi' \implies \text{wf_mformula } \sigma j P V n R \psi \psi' \implies$
 $\text{Formula.fv } \varphi' = \text{Formula.fv } \psi' \implies$
 $\text{wf_mbuf2}' \sigma P V j n R \varphi' \psi' \text{ buf} \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MOr } \varphi \psi \text{ buf}) (\text{Formula.Or } \varphi' \psi')$
| $\text{Neg: wf_mformula } \sigma j P V n R \varphi \varphi' \implies \text{Formula.fv } \varphi' = \{\} \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MNeg } \varphi) (\text{Formula.Neg } \varphi')$
| $\text{Exists: wf_mformula } \sigma j P V (\text{Suc } n) (\text{lift_envs } R) \varphi \varphi' \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MExists } \varphi) (\text{Formula.Exists } \varphi')$
| $\text{Agg: wf_mformula } \sigma j P V (b + n) (\text{lift_envs}' b R) \varphi \varphi' \implies$
 $y < n \implies$
 $y + b \notin \text{Formula.fv } \varphi' \implies$
 $\{0..<b\} \subseteq \text{Formula.fv } \varphi' \implies$
 $\text{Formula.fv_trm } f \subseteq \text{Formula.fv } \varphi' \implies$
 $g0 = (\text{Formula.fv } \varphi' \subseteq \{0..<b\}) \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MAgg } g0 y \omega b f \varphi) (\text{Formula.Agg } y \omega b f \varphi')$
| $\text{Prev: wf_mformula } \sigma j P V n R \varphi \varphi' \implies$
 $\text{first} \longleftrightarrow j = 0 \implies$
 $\text{list_all2 } (\lambda i. \text{qtable } n (\text{Formula.fv } \varphi') (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map } \text{the } v) i \varphi'))$
 $[\text{min } (\text{progress } \sigma P \varphi' j) (j-1)..<\text{progress } \sigma P \varphi' j] \text{ buf} \implies$
 $\text{list_all2 } (\lambda i t. t = \tau \sigma i) [\text{min } (\text{progress } \sigma P \varphi' j) (j-1)..<j] \text{ nts} \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MPrev } I \varphi \text{ first } \text{buf } \text{nts}) (\text{Formula.Prev } I \varphi')$
| $\text{Next: wf_mformula } \sigma j P V n R \varphi \varphi' \implies$
 $\text{first} \longleftrightarrow \text{progress } \sigma P \varphi' j = 0 \implies$
 $\text{list_all2 } (\lambda i t. t = \tau \sigma i) [\text{progress } \sigma P \varphi' j - 1..<j] \text{ nts} \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MNext } I \varphi \text{ first } \text{nts}) (\text{Formula.Next } I \varphi')$
| $\text{Since: wf_mformula } \sigma j P V n R \varphi \varphi' \implies \text{wf_mformula } \sigma j P V n R \psi \psi' \implies$
 $\text{if } \text{args_pos } \text{args} \text{ then } \varphi'' = \varphi' \text{ else } \varphi'' = \text{Formula.Neg } \varphi' \implies$
 $\text{safe_formula } \varphi'' = \text{args_pos } \text{args} \implies$
 $\text{args_ivl } \text{args} = I \implies$
 $\text{args_n } \text{args} = n \implies$
 $\text{args_L } \text{args} = \text{Formula.fv } \varphi' \implies$
 $\text{args_R } \text{args} = \text{Formula.fv } \psi' \implies$
 $\text{Formula.fv } \varphi' \subseteq \text{Formula.fv } \psi' \implies$
 $\text{wf_mbuf2}' \sigma P V j n R \varphi' \psi' \text{ buf} \implies$
 $\text{wf_ts } \sigma P j \varphi' \psi' \text{ nts} \implies$
 $\text{wf_since_aux } \sigma V R \text{args } \varphi' \psi' \text{aux } (\text{progress } \sigma P (\text{Formula.Since } \varphi'' I \psi') j) \implies$
 $\text{wf_mformula } \sigma j P V n R (\text{MSince } \text{args } \varphi \psi \text{buf } \text{nts } \text{aux}) (\text{Formula.Since } \varphi'' I \psi')$
| $\text{Until: wf_mformula } \sigma j P V n R \varphi \varphi' \implies \text{wf_mformula } \sigma j P V n R \psi \psi' \implies$
 $\text{if } \text{args_pos } \text{args} \text{ then } \varphi'' = \varphi' \text{ else } \varphi'' = \text{Formula.Neg } \varphi' \implies$

$safe_formula \varphi'' = args_pos \ args \implies$
 $args_ivl \ args = I \implies$
 $args_n \ args = n \implies$
 $args_L \ args = Formula.fv \ \varphi' \implies$
 $args_R \ args = Formula.fv \ \psi' \implies$
 $Formula.fv \ \varphi' \subseteq Formula.fv \ \psi' \implies$
 $wf_mbuf2' \ \sigma \ P \ V \ j \ n \ R \ \varphi' \ \psi' \ buf \implies$
 $wf_ts \ \sigma \ P \ j \ \varphi' \ \psi' \ nts \implies$
 $wf_until_aux \ \sigma \ V \ R \ args \ \varphi' \ \psi' \ aux \ (progress \ \sigma \ P \ (Formula.Until \ \varphi'' \ I \ \psi') \ j) \implies$
 $progress \ \sigma \ P \ (Formula.Until \ \varphi'' \ I \ \psi') \ j + length_muaux \ args \ aux = \min \ (progress \ \sigma \ P \ \varphi' \ j) \ (progress$
 $\sigma \ P \ \psi' \ j) \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MUntil \ args \ \varphi \ \psi \ buf \ nts \ aux) \ (Formula.Until \ \varphi'' \ I \ \psi')$
 $| \ MatchP: \ (case \ to_mregex \ r \ of \ (mr', \ \varphi s') \ \Rightarrow$
 $\quad list_all2 \ (wf_mformula \ \sigma \ j \ P \ V \ n \ R) \ \varphi s \ \varphi s' \ \wedge \ mr = mr') \ \implies$
 $mrs = sorted_list_of_set \ (RPDs \ mr) \ \implies$
 $safe_regex \ Past \ Strict \ r \ \implies$
 $wf_mbufn' \ \sigma \ P \ V \ j \ n \ R \ r \ buf \ \implies$
 $wf_ts_regex \ \sigma \ P \ j \ r \ nts \ \implies$
 $wf_matchP_aux \ \sigma \ V \ n \ R \ I \ r \ aux \ (progress \ \sigma \ P \ (Formula.MatchP \ I \ r) \ j) \ \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MMatchP \ I \ mr \ mrs \ \varphi s \ buf \ nts \ aux) \ (Formula.MatchP \ I \ r)$
 $| \ MatchF: \ (case \ to_mregex \ r \ of \ (mr', \ \varphi s') \ \Rightarrow$
 $\quad list_all2 \ (wf_mformula \ \sigma \ j \ P \ V \ n \ R) \ \varphi s \ \varphi s' \ \wedge \ mr = mr') \ \implies$
 $mrs = sorted_list_of_set \ (LPDs \ mr) \ \implies$
 $safe_regex \ Futu \ Strict \ r \ \implies$
 $wf_mbufn' \ \sigma \ P \ V \ j \ n \ R \ r \ buf \ \implies$
 $wf_ts_regex \ \sigma \ P \ j \ r \ nts \ \implies$
 $wf_matchF_aux \ \sigma \ V \ n \ R \ I \ r \ aux \ (progress \ \sigma \ P \ (Formula.MatchF \ I \ r) \ j) \ 0 \ \implies$
 $progress \ \sigma \ P \ (Formula.MatchF \ I \ r) \ j + length \ aux = progress_regex \ \sigma \ P \ r \ j \ \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MMatchF \ I \ mr \ mrs \ \varphi s \ buf \ nts \ aux) \ (Formula.MatchF \ I \ r)$

definition (in *maux*) $wf_mstate :: Formula.formula \Rightarrow Formula.prefix \Rightarrow event_data \ list \ set \Rightarrow ('msaux,$
 $'muaux) \ mstate \Rightarrow bool$ **where**

$wf_mstate \ \varphi \ \pi \ R \ st \ \longleftrightarrow \ mstate_n \ st = Formula.nfv \ \varphi \ \wedge \ (\forall \sigma. \ prefix_of \ \pi \ \sigma \ \longrightarrow$
 $\quad mstate_i \ st = progress \ \sigma \ Map.empty \ \varphi \ (plen \ \pi) \ \wedge$
 $\quad wf_mformula \ \sigma \ (plen \ \pi) \ Map.empty \ Map.empty \ (mstate_n \ st) \ R \ (mstate_m \ st) \ \varphi)$

6.6.2 Initialisation

lemma $wf_mbuf2'_0: \ pred_mapping \ (\lambda x. \ x = 0) \ P \ \implies \ wf_mbuf2' \ \sigma \ P \ V \ 0 \ n \ R \ \varphi \ \psi \ (\ [], \ [])$
 $\langle proof \rangle$

lemma $wf_mbufn'_0: \ to_mregex \ r = (mr, \ \varphi s) \ \implies \ pred_mapping \ (\lambda x. \ x = 0) \ P \ \implies \ wf_mbufn' \ \sigma \ P$
 $V \ 0 \ n \ R \ r \ (replicate \ (length \ \varphi s) \ [])$
 $\langle proof \rangle$

lemma $wf_ts_0: \ wf_ts \ \sigma \ P \ 0 \ \varphi \ \psi \ []$
 $\langle proof \rangle$

lemma $wf_ts_regex_0: \ wf_ts_regex \ \sigma \ P \ 0 \ r \ []$
 $\langle proof \rangle$

lemma (in *msaux*) $wf_since_aux_Nil: \ Formula.fv \ \varphi' \subseteq Formula.fv \ \psi' \ \implies$
 $wf_since_aux \ \sigma \ V \ R \ (init_args \ I \ n \ (Formula.fv \ \varphi') \ (Formula.fv \ \psi') \ b) \ \varphi' \ \psi' \ (init_msaux \ (init_args \ I$
 $n \ (Formula.fv \ \varphi') \ (Formula.fv \ \psi') \ b)) \ 0$
 $\langle proof \rangle$

lemma (in *muaux*) $wf_until_aux_Nil: \ Formula.fv \ \varphi' \subseteq Formula.fv \ \psi' \ \implies$
 $wf_until_aux \ \sigma \ V \ R \ (init_args \ I \ n \ (Formula.fv \ \varphi') \ (Formula.fv \ \psi') \ b) \ \varphi' \ \psi' \ (init_muaux \ (init_args \ I$
 $n \ (Formula.fv \ \varphi') \ (Formula.fv \ \psi') \ b)) \ 0$

<proof>

lemma *wf_matchP_aux_Nil*: *wf_matchP_aux* σ V n R I r $[]$ 0
<proof>

lemma *wf_matchF_aux_Nil*: *wf_matchF_aux* σ V n R I r $[]$ 0 k
<proof>

lemma *fv_regex_alt*: *safe_regex* m g r \implies *Formula.fv_regex* r = $(\bigcup \varphi \in \text{atms } r. \text{Formula.fv } \varphi)$
<proof>

lemmas *to_mregex_atms* =
to_mregex_ok[*THEN* *conjunct1*, *THEN* *equalityD1*, *THEN* *set_mp*, *rotated*]

lemma (*in mau*) *wf_minit0*: *safe_formula* φ \implies $\forall x \in \text{Formula.fv } \varphi. x < n \implies$
pred_mapping $(\lambda x. x = 0)$ $P \implies$
wf_mformula σ 0 P V n R (*minit0* n φ) φ
<proof>

lemma (*in mau*) *wf_mstate_minit*: *safe_formula* $\varphi \implies$ *wf_mstate* φ *pnil* R (*minit* φ)
<proof>

6.6.3 Evaluation

lemma *match_wf_tuple*: *Some* $f = \text{match } ts \ xs \implies$
wf_tuple n $(\bigcup t \in \text{set } ts. \text{Formula.fv_trm } t)$ $(\text{Table.tabulate } f \ 0 \ n)$
<proof>

lemma *match_fvi_trm_None*: *Some* $f = \text{match } ts \ xs \implies$ $\forall t \in \text{set } ts. x \notin \text{Formula.fv_trm } t \implies f \ x =$
None
<proof>

lemma *match_fvi_trm_Some*: *Some* $f = \text{match } ts \ xs \implies$ $t \in \text{set } ts \implies x \in \text{Formula.fv_trm } t \implies f \ x$
 $\neq \text{None}$
<proof>

lemma *match_eval_trm*: $\forall t \in \text{set } ts. \forall i \in \text{Formula.fv_trm } t. i < n \implies$ *Some* $f = \text{match } ts \ xs \implies$
map $(\text{Formula.eval_trm } (\text{Table.tabulate } (\lambda i. \text{the } (f \ i)) \ 0 \ n)) \ ts = xs$
<proof>

lemma *wf_tuple_tabulate_Some*: *wf_tuple* n A $(\text{Table.tabulate } f \ 0 \ n) \implies x \in A \implies x < n \implies \exists y. f \ x$
 $= \text{Some } y$
<proof>

lemma *ex_match*: *wf_tuple* n $(\bigcup t \in \text{set } ts. \text{Formula.fv_trm } t)$ $v \implies$
 $\forall t \in \text{set } ts. (\forall x \in \text{Formula.fv_trm } t. x < n) \wedge (\text{Formula.is_Var } t \vee \text{Formula.is_Const } t) \implies$
 $\exists f. \text{match } ts \ (\text{map } (\text{Formula.eval_trm } (\text{map the } v)) \ ts) = \text{Some } f \wedge v = \text{Table.tabulate } f \ 0 \ n$
<proof>

lemma *eq_rel_eval_trm*: $v \in \text{eq_rel } n \ t1 \ t2 \implies \text{is_simple_eq } t1 \ t2 \implies$
 $\forall x \in \text{Formula.fv_trm } t1 \cup \text{Formula.fv_trm } t2. x < n \implies$
Formula.eval_trm $(\text{map the } v) \ t1 = \text{Formula.eval_trm } (\text{map the } v) \ t2$
<proof>

lemma *in_eq_rel*: *wf_tuple* n $(\text{Formula.fv_trm } t1 \cup \text{Formula.fv_trm } t2)$ $v \implies$
is_simple_eq $t1 \ t2 \implies$
Formula.eval_trm $(\text{map the } v) \ t1 = \text{Formula.eval_trm } (\text{map the } v) \ t2 \implies$
 $v \in \text{eq_rel } n \ t1 \ t2$
<proof>

lemma *table_eq_rel: is_simple_eq t1 t2 \implies*
table n (Formula.fv_trm t1 \cup Formula.fv_trm t2) (eq_rel n t1 t2)
<proof>

lemma *wf_tuple_Suc_fviD: wf_tuple (Suc n) (Formula.fvi b φ) v \implies wf_tuple n (Formula.fvi (Suc b) φ) (tl v)*
<proof>

lemma *table_fvi_tl: table (Suc n) (Formula.fvi b φ) X \implies table n (Formula.fvi (Suc b) φ) (tl ' X)*
<proof>

lemma *wf_tuple_Suc_fvi_SomeI: 0 \in Formula.fvi b $\varphi \implies$ wf_tuple n (Formula.fvi (Suc b) φ) v \implies wf_tuple (Suc n) (Formula.fvi b φ) (Some x # v)*
<proof>

lemma *wf_tuple_Suc_fvi_NoneI: 0 \notin Formula.fvi b $\varphi \implies$ wf_tuple n (Formula.fvi (Suc b) φ) v \implies wf_tuple (Suc n) (Formula.fvi b φ) (None # v)*
<proof>

lemma *qtable_project_fv: qtable (Suc n) (fv φ) (mem_restr (lift_envs R)) P X \implies*
qtable n (Formula.fvi (Suc 0) φ) (mem_restr R)
($\lambda v. \exists x. P ((if 0 \in fv \varphi$ then Some x else None) # v)) (tl ' X)
<proof>

lemma *mem_restr_lift_envs'_append[simp]:*
length xs = b \implies mem_restr (lift_envs' b R) (xs @ ys) = mem_restr R ys
<proof>

lemma *nth_list_update_alt: xs[i := x] ! j = (if i < length xs \wedge i = j then x else xs ! j)*
<proof>

lemma *wf_tuple_upd_None: wf_tuple n A xs \implies A - {i} = B \implies wf_tuple n B (xs[i:=None])*
<proof>

lemma *mem_restr_upd_None: mem_restr R xs \implies mem_restr R (xs[i:=None])*
<proof>

lemma *mem_restr_dropI: mem_restr (lift_envs' b R) xs \implies mem_restr R (drop b xs)*
<proof>

lemma *mem_restr_dropD:*
assumes *b \leq length xs and mem_restr R (drop b xs)*
shows *mem_restr (lift_envs' b R) xs*
<proof>

lemma *wf_tuple_append: wf_tuple a {x \in A. x < a} xs \implies*
wf_tuple b {x - a | x. x \in A \wedge x \geq a} ys \implies
wf_tuple (a + b) A (xs @ ys)
<proof>

lemma *wf_tuple_map_Some: length xs = n \implies {0.. n } \subseteq A \implies wf_tuple n A (map Some xs)*
<proof>

lemma *wf_tuple_drop: wf_tuple (b + n) A xs \implies {x - b | x. x \in A \wedge x \geq b} = B \implies*
wf_tuple n B (drop b xs)
<proof>

lemma *ecard_image*: $\text{inj_on } f \ A \implies \text{ecard } (f \ ' \ A) = \text{ecard } A$
 ⟨proof⟩

lemma *meval_trm_eval_trm*: $\text{wf_tuple } n \ A \ x \implies \text{fv_trm } t \subseteq A \implies \forall i \in A. i < n \implies$
 $\text{meval_trm } t \ x = \text{Formula.eval_trm } (\text{map the } x) \ t$
 ⟨proof⟩

lemma *list_update_id*: $\text{xs } ! \ i = z \implies \text{xs}[i:=z] = \text{xs}$
 ⟨proof⟩

lemma *htable_wf_tupleD*: $\text{htable } n \ A \ P \ Q \ X \implies \forall x \in X. \text{wf_tuple } n \ A \ x$
 ⟨proof⟩

lemma *htable_eval_agg*:

assumes *inner*: $\text{htable } (b + n) \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } (\text{lift_envs}' \ b \ R))$

($\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi$) *rel*

and *n*: $\forall x \in \text{Formula.fv } (\text{Formula.Agg } y \ \omega \ b \ f \ \varphi). x < n$

and *fresh*: $y + b \notin \text{Formula.fv } \varphi$

and *b_fv*: $\{0..<b\} \subseteq \text{Formula.fv } \varphi$

and *f_fv*: $\text{Formula.fv_trm } f \subseteq \text{Formula.fv } \varphi$

and *g0*: $g0 = (\text{Formula.fv } \varphi \subseteq \{0..<b\})$

shows $\text{htable } n \ (\text{Formula.fv } (\text{Formula.Agg } y \ \omega \ b \ f \ \varphi)) \ (\text{mem_restr } R)$

($\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ (\text{Formula.Agg } y \ \omega \ b \ f \ \varphi)$) (*eval_agg* $n \ g0 \ y \ \omega \ b \ f \ \text{rel}$)

(**is** $\text{htable } _ \ ?\text{fv } _ \ ?Q \ ?\text{rel}'$)

⟨proof⟩

lemma *mprev*: $\text{mprev_next } I \ \text{xs } \ \text{ts} = (ys, xs', ts') \implies$

$\text{list_all2 } P \ [i..<j'] \ \text{xs} \implies \text{list_all2 } (\lambda i \ t. t = \tau \ \sigma \ i) \ [i..<j] \ \text{ts} \implies i \leq j' \implies i < j \implies$

$\text{list_all2 } (\lambda i \ X. \text{if mem } (\tau \ \sigma \ (\text{Suc } i) - \tau \ \sigma \ i) \ I \ \text{then } P \ i \ X \ \text{else } X = \text{empty_table})$

$[i..<\text{min } j' \ (j-1)] \ ys \ \wedge$

$\text{list_all2 } P \ [\text{min } j' \ (j-1)..<j'] \ \text{xs}' \ \wedge$

$\text{list_all2 } (\lambda i \ t. t = \tau \ \sigma \ i) \ [\text{min } j' \ (j-1)..<j] \ \text{ts}'$

⟨proof⟩

lemma *mnext*: $\text{mprev_next } I \ \text{xs } \ \text{ts} = (ys, xs', ts') \implies$

$\text{list_all2 } P \ [\text{Suc } i..<j'] \ \text{xs} \implies \text{list_all2 } (\lambda i \ t. t = \tau \ \sigma \ i) \ [i..<j] \ \text{ts} \implies \text{Suc } i \leq j' \implies i < j \implies$

$\text{list_all2 } (\lambda i \ X. \text{if mem } (\tau \ \sigma \ (\text{Suc } i) - \tau \ \sigma \ i) \ I \ \text{then } P \ (\text{Suc } i) \ X \ \text{else } X = \text{empty_table})$

$[i..<\text{min } (j'-1) \ (j-1)] \ ys \ \wedge$

$\text{list_all2 } P \ [\text{Suc } (\text{min } (j'-1) \ (j-1))..<j'] \ \text{xs}' \ \wedge$

$\text{list_all2 } (\lambda i \ t. t = \tau \ \sigma \ i) \ [\text{min } (j'-1) \ (j-1)..<j] \ \text{ts}'$

⟨proof⟩

lemma *in_foldr_UnI*: $x \in A \implies A \in \text{set } \text{xs} \implies x \in \text{foldr } (\cup) \ \text{xs } \{\}$

⟨proof⟩

lemma *in_foldr_UnE*: $x \in \text{foldr } (\cup) \ \text{xs } \{\} \implies (\bigwedge A. A \in \text{set } \text{xs} \implies x \in A \implies P) \implies P$

⟨proof⟩

lemma *sat_the_restrict*: $\text{fv } \varphi \subseteq A \implies \text{Formula.sat } \sigma \ V \ (\text{map the } (\text{restrict } A \ v)) \ i \ \varphi = \text{Formula.sat } \sigma$
 $V \ (\text{map the } v) \ i \ \varphi$

⟨proof⟩

lemma *eps_the_restrict*: $\text{fv_regex } r \subseteq A \implies \text{Regex.eps } (\text{Formula.sat } \sigma \ V \ (\text{map the } (\text{restrict } A \ v))) \ i \ r$
 $= \text{Regex.eps } (\text{Formula.sat } \sigma \ V \ (\text{map the } v)) \ i \ r$

⟨proof⟩

lemma *sorted_wrt_filter[simp]*: $\text{sorted_wrt } R \ \text{xs} \implies \text{sorted_wrt } R \ (\text{filter } P \ \text{xs})$

⟨proof⟩

lemma *concat_map_filter*[simp]:

concat (map f (filter P xs)) = concat (map ($\lambda x. \text{if } P \ x \ \text{then } f \ x \ \text{else } []$) xs)
 ⟨proof⟩

lemma *map_filter_alt*:

map f (filter P xs) = concat (map ($\lambda x. \text{if } P \ x \ \text{then } [f \ x] \ \text{else } []$) xs)
 ⟨proof⟩

lemma (in *maux*) *update_since*:

assumes *pre*: *wf_since_aux* $\sigma \ V \ R \ \text{args} \ \varphi \ \psi \ \text{aux} \ ne$

and *qtable1*: *qtable* $n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ ne \ \varphi) \ \text{rel1}$

and *qtable2*: *qtable* $n \ (\text{Formula.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ ne \ \psi) \ \text{rel2}$

and *result_eq*: $(\text{rel}, \text{aux}') = \text{update_since } \text{args } \text{rel1 } \text{rel2} \ (\tau \ \sigma \ ne) \ \text{aux}$

and *fv_subset*: *Formula.fv* $\varphi \subseteq \text{Formula.fv } \psi$

and *args_ivl*: *args_ivl* $\text{args} = I$

and *args_n*: *args_n* $\text{args} = n$

and *args_L*: *args_L* $\text{args} = \text{Formula.fv } \varphi$

and *args_R*: *args_R* $\text{args} = \text{Formula.fv } \psi$

and *args_pos*: *args_pos* $\text{args} = \text{pos}$

shows *wf_since_aux* $\sigma \ V \ R \ \text{args} \ \varphi \ \psi \ \text{aux}' \ (\text{Suc } ne)$

and *qtable* $n \ (\text{Formula.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ ne \ (\text{Sincep } \text{pos } \varphi \ I \ \psi)) \ \text{rel}$
 ⟨proof⟩

lemma *fv_regex_from_mregex*:

ok (length φs) mr $\implies \text{fv_regex } (\text{from_mregex } mr \ \varphi s) \subseteq (\bigcup \varphi \in \text{set } \varphi s. \text{fv } \varphi)$
 ⟨proof⟩

lemma *qtable_ε_lax*:

assumes *ok* $(\text{length } \varphi s) \ mr$

and *list_all2* $(\lambda \varphi \ \text{rel}. \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi) \ \text{rel}) \ \varphi s \ \text{rels}$

and *fv_regex* $(\text{from_mregex } mr \ \varphi s) \subseteq A$ **and** *qtable* $n \ A \ (\text{mem_restr } R) \ Q \ \text{guard}$

shows *qtable* $n \ A \ (\text{mem_restr } R)$

$(\lambda v. \text{Regex.eps } (\text{Formula.sat } \sigma \ V \ (\text{map the } v)) \ i \ (\text{from_mregex } mr \ \varphi s) \wedge Q \ v) \ (\varepsilon_lax \ \text{guard} \ \text{rels } mr)$

⟨proof⟩

lemma *nullary_qtable_cases*: *qtable* $n \ \{\} \ P \ Q \ X \implies (X = \text{empty_table} \vee X = \text{unit_table } n)$

⟨proof⟩

lemma *qtable_empty_unit_table*:

qtable $n \ \{\} \ R \ P \ \text{empty_table} \implies \text{qtable } n \ \{\} \ R \ (\lambda v. \neg P \ v) \ (\text{unit_table } n)$

⟨proof⟩

lemma *qtable_unit_empty_table*:

qtable $n \ \{\} \ R \ P \ (\text{unit_table } n) \implies \text{qtable } n \ \{\} \ R \ (\lambda v. \neg P \ v) \ \text{empty_table}$

⟨proof⟩

lemma *qtable_nonempty_empty_table*:

qtable $n \ \{\} \ R \ P \ X \implies x \in X \implies \text{qtable } n \ \{\} \ R \ (\lambda v. \neg P \ v) \ \text{empty_table}$

⟨proof⟩

lemma *qtable_rε_strict*:

assumes *safe_regex Past Strict* $(\text{from_mregex } mr \ \varphi s) \ \text{ok} \ (\text{length } \varphi s) \ mr \ A = \text{fv_regex } (\text{from_mregex } mr \ \varphi s)$

and *list_all2* $(\lambda \varphi \ \text{rel}. \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i$

φ) *rel*) φ s *rels*
shows *qtable* n A (*mem_restr* R) (λv . *Regex.eps* (*Formula.sat* σ V (*map the v*)) i (*from_mregex* mr φ s)) (*re_strict* n *rels* mr)
 ⟨*proof*⟩

lemma *qtable_lε_strict*:

assumes *safe_regex* *Futu Strict* (*from_mregex* mr φ s) *ok* (*length* φ s) mr $A = fv_regex$ (*from_mregex* mr φ s)
and *list_all2* ($\lambda\varphi$ *rel*. *qtable* n (*Formula.fv* φ) (*mem_restr* R) (λv . *Formula.sat* σ V (*map the v*) i φ) *rel*) φ s *rels*
shows *qtable* n A (*mem_restr* R) (λv . *Regex.eps* (*Formula.sat* σ V (*map the v*)) i (*from_mregex* mr φ s)) (*lε_strict* n *rels* mr)
 ⟨*proof*⟩

lemma *rtranclp_False*: (λi j . *False*)^{**} = (=)
 ⟨*proof*⟩

inductive *ok_rctxt* **for** φ s **where**

ok_rctxt φ s *id id*
 | *ok_rctxt* φ s κ κ' $\implies ok_rctxt$ φ s (λt . κ (*MTimes* mr t)) (λt . κ' (*Regex.Times* (*from_mregex* mr φ s) t))

lemma *ok_rctxt_swap*: *ok_rctxt* φ s κ κ' $\implies from_mregex$ (κ mr) φ s = κ' (*from_mregex* mr φ s)
 ⟨*proof*⟩

lemma *ok_rctxt_cong*: *ok_rctxt* φ s κ κ' $\implies Regex.match$ (*Formula.sat* σ V v) $r = Regex.match$ (*Formula.sat* σ V v) $s \implies$
 $Regex.match$ (*Formula.sat* σ V v) (κ' r) i $j = Regex.match$ (*Formula.sat* σ V v) (κ' s) i j
 ⟨*proof*⟩

lemma *qtable_rδκ*:

assumes *ok* (*length* φ s) mr *fv_regex* (*from_mregex* mr φ s) $\subseteq A$
and *list_all2* ($\lambda\varphi$ *rel*. *qtable* n (*Formula.fv* φ) (*mem_restr* R) (λv . *Formula.sat* σ V (*map the v*) j φ) *rel*) φ s *rels*
and *ok_rctxt* φ s κ κ'
and $\forall ms \in \kappa' \text{ RPD } mr$. *qtable* n A (*mem_restr* R) (λv . Q (*map the v*) (*from_mregex* ms φ s)) (*lookup* *rel* ms)
shows *qtable* n A (*mem_restr* R)
 (λv . $\exists s \in Regex.rpd\kappa$ κ' (*Formula.sat* σ V (*map the v*)) j (*from_mregex* mr φ s)). Q (*map the v*) s)
 (*rδ* κ *rel* *rels* mr)
 ⟨*proof*⟩

lemmas *qtable_rδ* = *qtable_rδκ*[*OF* _ _ _ *ok_rctxt.intros*(1), *unfolded* *rpdκ_rpd* *image_id* *id_apply*]

inductive *ok_lctxt* **for** φ s **where**

ok_lctxt φ s *id id*
 | *ok_lctxt* φ s κ κ' $\implies ok_lctxt$ φ s (λt . κ (*MTimes* t mr)) (λt . κ' (*Regex.Times* t (*from_mregex* mr φ s)))

lemma *ok_lctxt_swap*: *ok_lctxt* φ s κ κ' $\implies from_mregex$ (κ mr) φ s = κ' (*from_mregex* mr φ s)
 ⟨*proof*⟩

lemma *ok_lctxt_cong*: *ok_lctxt* φ s κ κ' $\implies Regex.match$ (*Formula.sat* σ V v) $r = Regex.match$ (*Formula.sat* σ V v) $s \implies$
 $Regex.match$ (*Formula.sat* σ V v) (κ' r) i $j = Regex.match$ (*Formula.sat* σ V v) (κ' s) i j
 ⟨*proof*⟩

lemma *qtable_lδκ*:

assumes *ok* (*length* φ s) mr *fv_regex* (*from_mregex* mr φ s) $\subseteq A$

and *list_all2* ($\lambda\varphi$ *rel. qtable n (Formula.fv φ) (mem_restr R) ($\lambda v. Formula.sat \sigma V (map\ the\ v) j$)* *rel*) *φs rels*
and *ok_lctxt* *$\varphi s \kappa \kappa'$*
and $\forall ms \in \kappa$ ' *LPD mr. qtable n A (mem_restr R) ($\lambda v. Q (map\ the\ v) (from_mregex\ ms\ \varphi s)$) (lookup rel ms)*
shows *qtable n A (mem_restr R)*
 $(\lambda v. \exists s \in Regex.lpd\kappa\ \kappa' (Formula.sat\ \sigma\ V\ (map\ the\ v))\ j\ (from_mregex\ mr\ \varphi s). Q\ (map\ the\ v)\ s)$
 $(l\delta\ \kappa\ rel\ rels\ mr)$
<proof>

lemmas *qtable_l δ = qtable_l δ κ [OF _ _ _ ok_lctxt.intros(1), unfolded lpd κ _lpd image_id id_apply]*

lemma *RPD_fv_regex_le:*
 $ms \in RPD\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) \subseteq fv_regex\ (from_mregex\ mr\ \varphi s)$
<proof>

lemma *RPD_safe: safe_regex Past g (from_mregex mr φs) \implies*
 $ms \in RPD\ mr \implies safe_regex\ Past\ g\ (from_mregex\ ms\ \varphi s)$
<proof>

lemma *RPDi_safe: safe_regex Past g (from_mregex mr φs) \implies*
 $ms \in RPDi\ n\ mr \implies safe_regex\ Past\ g\ (from_mregex\ ms\ \varphi s)$
<proof>

lemma *RPDs_safe: safe_regex Past g (from_mregex mr φs) \implies*
 $ms \in RPDs\ mr \implies safe_regex\ Past\ g\ (from_mregex\ ms\ \varphi s)$
<proof>

lemma *RPD_safe_fv_regex: safe_regex Past Strict (from_mregex mr φs) \implies*
 $ms \in RPD\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
<proof>

lemma *RPDi_safe_fv_regex: safe_regex Past Strict (from_mregex mr φs) \implies*
 $ms \in RPDi\ n\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
<proof>

lemma *RPDs_safe_fv_regex: safe_regex Past Strict (from_mregex mr φs) \implies*
 $ms \in RPDs\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
<proof>

lemma *RPD_ok: ok m mr \implies ms \in RPD mr \implies ok m ms*
<proof>

lemma *RPDi_ok: ok m mr \implies ms \in RPDi n mr \implies ok m ms*
<proof>

lemma *RPDs_ok: ok m mr \implies ms \in RPDs mr \implies ok m ms*
<proof>

lemma *LPD_fv_regex_le:*
 $ms \in LPD\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) \subseteq fv_regex\ (from_mregex\ mr\ \varphi s)$
<proof>

lemma *LPD_safe: safe_regex Futu g (from_mregex mr φs) \implies*
 $ms \in LPD\ mr \implies safe_regex\ Futu\ g\ (from_mregex\ ms\ \varphi s)$
<proof>

lemma *LPDi_safe: safe_regex Futu g (from_mregex mr φs) \implies*

$ms \in LPDi\ n\ mr \implies safe_regex\ Futu\ g\ (from_mregex\ ms\ \varphi s)$
 ⟨proof⟩

lemma $LPDs_safe$: $safe_regex\ Futu\ g\ (from_mregex\ mr\ \varphi s) \implies$
 $ms \in LPDs\ mr \implies safe_regex\ Futu\ g\ (from_mregex\ ms\ \varphi s)$
 ⟨proof⟩

lemma $LPD_safe_fv_regex$: $safe_regex\ Futu\ Strict\ (from_mregex\ mr\ \varphi s) \implies$
 $ms \in LPD\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
 ⟨proof⟩

lemma $LPDi_safe_fv_regex$: $safe_regex\ Futu\ Strict\ (from_mregex\ mr\ \varphi s) \implies$
 $ms \in LPDi\ n\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
 ⟨proof⟩

lemma $LPDs_safe_fv_regex$: $safe_regex\ Futu\ Strict\ (from_mregex\ mr\ \varphi s) \implies$
 $ms \in LPDs\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
 ⟨proof⟩

lemma LPD_ok : $ok\ m\ mr \implies ms \in LPD\ mr \implies ok\ m\ ms$
 ⟨proof⟩

lemma $LPDi_ok$: $ok\ m\ mr \implies ms \in LPDi\ n\ mr \implies ok\ m\ ms$
 ⟨proof⟩

lemma $LPDs_ok$: $ok\ m\ mr \implies ms \in LPDs\ mr \implies ok\ m\ ms$
 ⟨proof⟩

lemma $update_matchP$:

assumes pre : $wf_matchP_aux\ \sigma\ V\ n\ R\ I\ r\ aux\ ne$
and $safe$: $safe_regex\ Past\ Strict\ r$
and mr : $to_mregex\ r = (mr, \varphi s)$
and mrs : $mrs = sorted_list_of_set\ (RPDs\ mr)$
and $qtables$: $list_all2\ (\lambda\varphi\ rel.\ qtable\ n\ (Formula.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ ne\ \varphi)\ rel)\ \varphi s\ rels$
and $result_eq$: $(rel, aux') = update_matchP\ n\ I\ mr\ mrs\ rels\ (\tau\ \sigma\ ne)\ aux$
shows $wf_matchP_aux\ \sigma\ V\ n\ R\ I\ r\ aux'\ (Suc\ ne)$
and $qtable\ n\ (Formula.fv_regex\ r)\ (mem_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ ne\ (Formula.MatchP\ I\ r))\ rel$
 ⟨proof⟩

lemma $length_update_until$: $length\ (update_until\ args\ rel1\ rel2\ nt\ aux) = Suc\ (length\ aux)$
 ⟨proof⟩

lemma $wf_update_until_auxlist$:

assumes pre : $wf_until_auxlist\ \sigma\ V\ n\ R\ pos\ \varphi\ I\ \psi\ auxlist\ ne$
and $qtable1$: $qtable\ n\ (Formula.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ (ne + length\ auxlist)\ \varphi)\ rel1$
and $qtable2$: $qtable\ n\ (Formula.fv\ \psi)\ (mem_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ (ne + length\ auxlist)\ \psi)\ rel2$
and fv_subset : $Formula.fv\ \varphi \subseteq Formula.fv\ \psi$
and $args_ivl$: $args_ivl\ args = I$
and $args_n$: $args_n\ args = n$
and $args_pos$: $args_pos\ args = pos$
shows $wf_until_auxlist\ \sigma\ V\ n\ R\ pos\ \varphi\ I\ \psi\ (update_until\ args\ rel1\ rel2\ (\tau\ \sigma\ (ne + length\ auxlist)))\ auxlist\ ne$
 ⟨proof⟩

lemma (in *muaux*) *wf_update_until*:
assumes *pre*: *wf_until_aux* σ V R *args* φ ψ *aux* *ne*
and *qtable1*: *qtable* n (*Formula.fv* φ) (*mem_restr* R) (λv . *Formula.sat* σ V (*map the v*) ($ne + \text{length_muaux } args \text{ aux}$) φ) *rel1*
and *qtable2*: *qtable* n (*Formula.fv* ψ) (*mem_restr* R) (λv . *Formula.sat* σ V (*map the v*) ($ne + \text{length_muaux } args \text{ aux}$) ψ) *rel2*
and *fvi_subset*: *Formula.fv* $\varphi \subseteq$ *Formula.fv* ψ
and *args_ivl*: *args_ivl* *args* = I
and *args_n*: *args_n* *args* = n
and *args_L*: *args_L* *args* = *Formula.fv* φ
and *args_R*: *args_R* *args* = *Formula.fv* ψ
and *args_pos*: *args_pos* *args* = *pos*
shows *wf_until_aux* σ V R *args* φ ψ (*add_new_muaux* *args* *rel1* *rel2* (τ σ ($ne + \text{length_muaux } args \text{ aux}$)) *aux*) $ne \wedge$
 $\text{length_muaux } args$ (*add_new_muaux* *args* *rel1* *rel2* (τ σ ($ne + \text{length_muaux } args \text{ aux}$)) *aux*) =
Suc ($\text{length_muaux } args \text{ aux}$)
<proof>

lemma *length_update_matchF_base*:
 length (*fst* (*update_matchF_base* I *mr* *mrs* *nt* *entry* *st*)) = *Suc* 0
<proof>

lemma *length_update_matchF_step*:
 length (*fst* (*update_matchF_step* I *mr* *mrs* *nt* *entry* *st*)) = *Suc* (length (*fst* *st*))
<proof>

lemma *length_foldr_update_matchF_step*:
 length (*fst* (*foldr* (*update_matchF_step* I *mr* *mrs* *nt*) *aux* *base*)) = $\text{length } aux + \text{length}$ (*fst* *base*)
<proof>

lemma *length_update_matchF*: length (*update_matchF* n I *mr* *mrs* *rels* *nt* *aux*) = *Suc* ($\text{length } aux$)
<proof>

lemma *wf_update_matchF_base_invar*:
assumes *safe*: *safe_regex* *Futu* *Strict* r
and *mr*: *to_mregex* $r = (mr, \varphi s)$
and *mrs*: *mrs* = *sorted_list_of_set* (*LPDs* *mr*)
and *qtables*: *list_all2* ($\lambda \varphi$ *rel*. *qtable* n (*Formula.fv* φ) (*mem_restr* R) (λv . *Formula.sat* σ V (*map the v*) j φ) *rel*) φs *rels*
shows *wf_matchF_invar* σ V n R I r (*update_matchF_base* n I *mr* *mrs* *rels* (τ σ j)) j
<proof>

lemma *Un_empty_table[simp]*: $rel \cup \text{empty_table} = rel \text{ empty_table} \cup rel = rel$
<proof>

lemma *wf_matchF_invar_step*:
assumes *wf*: *wf_matchF_invar* σ V n R I r *st* (*Suc* i)
and *safe*: *safe_regex* *Futu* *Strict* r
and *mr*: *to_mregex* $r = (mr, \varphi s)$
and *mrs*: *mrs* = *sorted_list_of_set* (*LPDs* *mr*)
and *qtables*: *list_all2* ($\lambda \varphi$ *rel*. *qtable* n (*Formula.fv* φ) (*mem_restr* R) (λv . *Formula.sat* σ V (*map the v*) i φ) *rel*) φs *rels*
and *rel*: *qtable* n (*Formula.fv_regex* r) (*mem_restr* R) (λv . ($\exists j$. $i \leq j \wedge j < i + \text{length}$ (*fst* *st*) \wedge *mem* (τ σ $j - \tau$ σ i) I \wedge *Regex.match* (*Formula.sat* σ V (*map the v*)) r i j)) *rel*
and *entry*: *entry* = (τ σ i , *rels*, *rel*)
and *nt*: *nt* = τ σ ($i + \text{length}$ (*fst* *st*))
shows *wf_matchF_invar* σ V n R I r (*update_matchF_step* I *mr* *mrs* *nt* *entry* *st*) i

<proof>

lemma *wf_update_matchF_invar*:

assumes *pre*: *wf_matchF_aux* σ V n R I r *aux* ne (*length* (*fst st*) - 1)
and *wf*: *wf_matchF_invar* σ V n R I r *st* ($ne + \text{length } aux$)
and *safe*: *safe_regex Futu Strict* r
and *mr*: *to_mregex* $r = (mr, \varphi s)$
and *mrs*: *mrs = sorted_list_of_set* (*LPDs* mr)
and *j*: $j = ne + \text{length } aux + \text{length } (fst st) - 1$
shows *wf_matchF_invar* σ V n R I r (*foldr* (*update_matchF_step* I mr mrs ($\tau \sigma j$)) *aux st*) ne
<proof>

lemma *wf_update_matchF*:

assumes *pre*: *wf_matchF_aux* σ V n R I r *aux* ne 0
and *safe*: *safe_regex Futu Strict* r
and *mr*: *to_mregex* $r = (mr, \varphi s)$
and *mrs*: *mrs = sorted_list_of_set* (*LPDs* mr)
and *nt*: $nt = \tau \sigma (ne + \text{length } aux)$
and *qtables*: *list_all2* ($\lambda \varphi \text{ rel. } qtable$ n (*Formula.fv* φ) (*mem_restr* R) ($\lambda v. \text{Formula.sat } \sigma$ V (*map the v*) ($ne + \text{length } aux$) φ) *rel*) φs *rels*
shows *wf_matchF_aux* σ V n R I r (*update_matchF* n I mr mrs *rels* nt *aux*) ne 0
<proof>

lemma *wf_until_aux_Cons*: *wf_until_auxlist* σ V n R *pos* φ I ψ ($a \# aux$) $ne \implies$
wf_until_auxlist σ V n R *pos* φ I ψ *aux* (*Suc* ne)
<proof>

lemma *wf_matchF_aux_Cons*: *wf_matchF_aux* σ V n R I r (*entry* $\# aux$) ne $i \implies$
wf_matchF_aux σ V n R I r *aux* (*Suc* ne) i
<proof>

lemma *wf_until_aux_Cons1*: *wf_until_auxlist* σ V n R *pos* φ I ψ ($(t, a1, a2) \# aux$) $ne \implies t = \tau \sigma$
 ne
<proof>

lemma *wf_matchF_aux_Cons1*: *wf_matchF_aux* σ V n R I r ($(t, \text{rels}, \text{rel}) \# aux$) ne $i \implies t = \tau \sigma$
 ne
<proof>

lemma *wf_until_aux_Cons3*: *wf_until_auxlist* σ V n R *pos* φ I ψ ($(t, a1, a2) \# aux$) $ne \implies$
qtable n (*Formula.fv* ψ) (*mem_restr* R) ($\lambda v. (\exists j. ne \leq j \wedge j < \text{Suc } (ne + \text{length } aux) \wedge \text{mem } (\tau \sigma j - \tau \sigma ne) I \wedge$
 $\text{Formula.sat } \sigma$ V (*map the v*) $j \psi \wedge (\forall k \in \{ne..<j\}. \text{if } pos \text{ then } \text{Formula.sat } \sigma$ V (*map the v*) $k \varphi$ else
 $\neg \text{Formula.sat } \sigma$ V (*map the v*) $k \varphi))$) $a2$
<proof>

lemma *wf_matchF_aux_Cons3*: *wf_matchF_aux* σ V n R I r ($(t, \text{rels}, \text{rel}) \# aux$) ne $i \implies$
qtable n (*Formula.fv_regex* r) (*mem_restr* R) ($\lambda v. (\exists j. ne \leq j \wedge j < \text{Suc } (ne + \text{length } aux + i) \wedge \text{mem } (\tau \sigma j - \tau \sigma ne) I \wedge$
 $\text{Regex.match } (\text{Formula.sat } \sigma$ V (*map the v*)) r ne j)) *rel*
<proof>

lemma *upt_append*: $a \leq b \implies b \leq c \implies [a..<b] @ [b..<c] = [a..<c]$
<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 $ja \leq ja'\ jb \leq jb'$
shows $wf_mbuf2\ i\ ja'\ jb'\ P\ Q\ (mbuf2_add\ xs\ ys\ buf)$
 $\langle proof \rangle$

lemma wf_mbufn_add :
assumes $wf_mbufn\ i\ js\ Ps\ buf$
and $list_all3\ list_all2\ Ps\ (List.map2\ (\lambda j\ j'.\ [j..<j'])\ js\ js')\ xss$
and $list_all2\ (\leq)\ js\ js'$
shows $wf_mbufn\ i\ js'\ Ps\ (mbufn_add\ xss\ buf)$
 $\langle proof \rangle$

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 $list_all2\ (\lambda i\ z.\ \exists x\ y.\ P\ i\ x \wedge Q\ i\ y \wedge z = f\ x\ y)\ [i..<min\ ja\ jb]\ xs$
 $\langle proof \rangle$

lemma $list_all3_Nil[simp]$:
 $list_all3\ P\ xs\ ys\ [] \longleftrightarrow xs = [] \wedge ys = []$
 $list_all3\ P\ xs\ []\ zs \longleftrightarrow xs = [] \wedge zs = []$
 $list_all3\ P\ []\ ys\ zs \longleftrightarrow ys = [] \wedge zs = []$
 $\langle proof \rangle$

lemma $list_all3_Cons$:
 $list_all3\ P\ xs\ ys\ (z\ \#\ zs) \longleftrightarrow (\exists x\ xs'\ y\ ys'.\ xs = x\ \#\ xs' \wedge ys = y\ \#\ ys' \wedge P\ x\ y\ z \wedge list_all3\ P\ xs'\ ys'\ zs)$
 $list_all3\ P\ xs\ (y\ \#\ ys)\ zs \longleftrightarrow (\exists x\ xs'\ z\ zs'.\ xs = x\ \#\ xs' \wedge zs = z\ \#\ zs' \wedge P\ x\ y\ z \wedge list_all3\ P\ xs'\ ys'\ zs')$
 $list_all3\ P\ (x\ \#\ xs)\ ys\ zs \longleftrightarrow (\exists y\ ys'\ z\ zs'.\ ys = y\ \#\ ys' \wedge zs = z\ \#\ zs' \wedge P\ x\ y\ z \wedge list_all3\ P\ xs\ ys'\ zs')$
 $\langle proof \rangle$

lemma $list_all3_mono_strong$: $list_all3\ P\ xs\ ys\ zs \implies$
 $(\bigwedge x\ y\ z.\ x \in set\ xs \implies y \in set\ ys \implies z \in set\ zs \implies P\ x\ y\ z \implies Q\ x\ y\ z) \implies$
 $list_all3\ Q\ xs\ ys\ zs$
 $\langle proof \rangle$

definition $Mini\ where$
 $Mini\ i\ js = (if\ js = []\ then\ i\ else\ Min\ (set\ js))$

lemma $wf_mbufn_in_set_Mini$:
assumes $wf_mbufn\ i\ js\ Ps\ buf$
shows $[] \in set\ buf \implies Mini\ i\ js = i$
 $\langle proof \rangle$

lemma $wf_mbufn_notin_set$:
assumes $wf_mbufn\ i\ js\ Ps\ buf$
shows $[] \notin set\ buf \implies j \in set\ js \implies i < j$
 $\langle proof \rangle$

lemma $wf_mbufn_map_tl$:
 $wf_mbufn\ i\ js\ Ps\ buf \implies [] \notin set\ buf \implies wf_mbufn\ (Suc\ i)\ js\ Ps\ (map\ tl\ buf)$
 $\langle proof \rangle$

lemma $list_all3_list_all2I$: $list_all3\ (\lambda x\ y\ z.\ Q\ x\ z)\ xs\ ys\ zs \implies list_all2\ Q\ xs\ zs$

<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 $ja \leq j \text{ } jb \leq j$
shows *wf_mbuf2* $(\min ja jb) ja jb P Q buf'$
and *list_all2* $R [\min ja jb..<j] nts'$
<proof>

lemma *wf_mbufn_take*:

assumes *mbufn_take* $f z buf = (z', buf')$
and *wf_mbufn* $i js Ps buf$
shows *wf_mbufn* $(\text{Mini } i js) js Ps buf'$
<proof>

lemma *mbufnt_take_eqD*:

assumes *mbufnt_take* $f z buf nts = (z', buf', nts')$
and *wf_mbufn* $i js Ps buf$
and *list_all2* $R [i..<j] nts$
and $\bigwedge k. k \in \text{set } js \implies k \leq j$
and $k = \text{Mini } (i + \text{length } nts) js$
shows *wf_mbufn* $k js Ps buf'$
and *list_all2* $R [k..<j] nts'$
<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 $ja \leq j \text{ } jb \leq j$
and $U i z$
and $\bigwedge k x y t z. i \leq k \implies \text{Suc } k \leq ja \implies \text{Suc } k \leq jb \implies$
 $P k x \implies Q k y \implies R k t \implies U k z \implies U (\text{Suc } k) (f x y t z)$
shows $U (\min ja jb) z'$
<proof>

lemma *list_all2_hdD*:

assumes *list_all2* $P [i..<j] xs xs \neq []$
shows $P i (\text{hd } xs) i < j$
<proof>

lemma *mbufn_take_induct*[*consumes 3, case_names base step*]:

assumes *mbufn_take* $f z buf = (z', buf')$
and *wf_mbufn* $i js Ps buf$
and $U i z$
and $\bigwedge k xs z. i \leq k \implies \text{Suc } k \leq \text{Mini } i js \implies$
 $\text{list_all2 } (\lambda P x. P k x) Ps xs \implies U k z \implies U (\text{Suc } k) (f xs z)$
shows $U (\text{Mini } i js) z'$
<proof>

lemma *mbufnt_take_induct*[*consumes 5, case_names base step*]:

assumes *mbufnt_take* $f z buf nts = (z', buf', nts')$
and *wf_mbufn* $i js Ps buf$
and *list_all2* $R [i..<j] nts$
and $\bigwedge k. k \in \text{set } js \implies k \leq j$
and $U i z$

and $\bigwedge k \ xs \ t \ z. \ i \leq k \implies \text{Suc } k \leq \text{Mini } j \ j's \implies$
 $\text{list_all2 } (\lambda P \ x. \ P \ k \ x) \ P's \ xs \implies R \ k \ t \implies U \ k \ z \implies U \ (\text{Suc } k) \ (f \ xs \ t \ z)$
shows $U \ (\text{Mini } (i + \text{length } nts) \ j's) \ z'$
 <proof>

lemma *mbuf2_take_add'*:

assumes *eq*: $\text{mbuf2_take } f \ (\text{mbuf2_add } xs \ ys \ buf) = (zs, \text{buf}')$
and *pre*: $\text{wf_mbuf2}' \ \sigma \ P \ V \ j \ n \ R \ \varphi \ \psi \ buf$
and *rm*: $\text{rel_mapping } (\leq) \ P \ P'$
and *xs*: $\text{list_all2 } (\lambda i. \ \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \varphi))$
 $[\text{progress } \sigma \ P \ \varphi \ j..<\text{progress } \sigma \ P' \ \varphi \ j'] \ xs$
and *ys*: $\text{list_all2 } (\lambda i. \ \text{qtable } n \ (\text{Formula.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \psi))$
 $[\text{progress } \sigma \ P \ \psi \ j..<\text{progress } \sigma \ P' \ \psi \ j'] \ ys$
and $j \leq j'$
shows $\text{wf_mbuf2}' \ \sigma \ P' \ V \ j' \ n \ R \ \varphi \ \psi \ \text{buf}'$
and $\text{list_all2 } (\lambda i \ Z. \ \exists X \ Y. \ \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \varphi) \ X \wedge$
 $\text{qtable } n \ (\text{Formula.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \psi) \ Y \wedge$
 $Z = f \ X \ Y)$
 $[\text{min } (\text{progress } \sigma \ P \ \varphi \ j) \ (\text{progress } \sigma \ P \ \psi \ j)..<\text{min } (\text{progress } \sigma \ P' \ \varphi \ j') \ (\text{progress } \sigma \ P' \ \psi \ j')]] \ zs$
 <proof>

lemma *mbuf2t_take_add'*:

assumes *eq*: $\text{mbuf2t_take } f \ z \ (\text{mbuf2_add } xs \ ys \ buf) \ nts = (z', \text{buf}', \text{nts}')$
and *bounded*: $\text{pred_mapping } (\lambda x. \ x \leq j) \ P \ \text{pred_mapping } (\lambda x. \ x \leq j') \ P'$
and *rm*: $\text{rel_mapping } (\leq) \ P \ P'$
and *pre_buf*: $\text{wf_mbuf2}' \ \sigma \ P \ V \ j \ n \ R \ \varphi \ \psi \ buf$
and *pre_nts*: $\text{list_all2 } (\lambda i \ t. \ t = \tau \ \sigma \ i) \ [\text{min } (\text{progress } \sigma \ P \ \varphi \ j) \ (\text{progress } \sigma \ P \ \psi \ j)..<j'] \ nts$
and *xs*: $\text{list_all2 } (\lambda i. \ \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \varphi))$
 $[\text{progress } \sigma \ P \ \varphi \ j..<\text{progress } \sigma \ P' \ \varphi \ j'] \ xs$
and *ys*: $\text{list_all2 } (\lambda i. \ \text{qtable } n \ (\text{Formula.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \psi))$
 $[\text{progress } \sigma \ P \ \psi \ j..<\text{progress } \sigma \ P' \ \psi \ j'] \ ys$
and $j \leq j'$
shows $\text{wf_mbuf2}' \ \sigma \ P' \ V \ j' \ n \ R \ \varphi \ \psi \ \text{buf}'$
and $\text{wf_ts } \sigma \ P' \ j' \ \varphi \ \psi \ \text{nts}'$
 <proof>

lemma *ok_0_atms*: $\text{ok } 0 \ mr \implies \text{regex.atms } (\text{from_mregex } mr \ []) = \{\}$
 <proof>

lemma *ok_0_progress*: $\text{ok } 0 \ mr \implies \text{progress_regex } \sigma \ P \ (\text{from_mregex } mr \ []) \ j = j$
 <proof>

lemma *atms_empty_atms*: $\text{safe_regex } m \ g \ r \implies \text{atms } r = \{\} \longleftrightarrow \text{regex.atms } r = \{\}$
 <proof>

lemma *atms_empty_progress*: $\text{safe_regex } m \ g \ r \implies \text{atms } r = \{\} \implies \text{progress_regex } \sigma \ P \ r \ j = j$
 <proof>

lemma *to_mregex_empty_progress*: $\text{safe_regex } m \ g \ r \implies \text{to_mregex } r = (mr, []) \implies$
 $\text{progress_regex } \sigma \ P \ r \ j = j$
 <proof>

lemma *progress_regex_le*: $\text{pred_mapping } (\lambda x. \ x \leq j) \ P \implies \text{progress_regex } \sigma \ P \ r \ j \leq j$
 <proof>

lemma *Neg_acyclic*: $\text{formula.Neg } x = x \implies P$
 <proof>

lemma *case_Neg_in_iff*: $x \in (\text{case } y \text{ of formula.Neg } \varphi' \Rightarrow \{\varphi'\} \mid _ \Rightarrow \{\}) \iff y = \text{formula.Neg } x$
 ⟨proof⟩

lemma *atms_nonempty_progress*:

safe_regex m g $r \implies \text{atms } r \neq \{\}$ $\implies (\lambda\varphi. \text{progress } \sigma P \varphi j) \text{ 'atms } r = (\lambda\varphi. \text{progress } \sigma P \varphi j) \text{ '}$
regex.atms r
 ⟨proof⟩

lemma *to_mregex_nonempty_progress*: *safe_regex* m g $r \implies \text{to_mregex } r = (mr, \varphi s) \implies \varphi s \neq [] \implies$
progress_regex $\sigma P r j = (\text{MIN } \varphi \in \text{set } \varphi s. \text{progress } \sigma P \varphi j)$
 ⟨proof⟩

lemma *to_mregex_progress*: *safe_regex* m g $r \implies \text{to_mregex } r = (mr, \varphi s) \implies$
progress_regex $\sigma P r j = (\text{if } \varphi s = [] \text{ then } j \text{ else } (\text{MIN } \varphi \in \text{set } \varphi s. \text{progress } \sigma P \varphi j))$
 ⟨proof⟩

lemma *mbufnt_take_add'*:

assumes *eq*: *mbufnt_take* f z (*mbufn_add* xss buf) $nts = (z', buf', nts')$
and *bounded*: *pred_mapping* $(\lambda x. x \leq j)$ P *pred_mapping* $(\lambda x. x \leq j')$ P'
and *rm*: *rel_mapping* (\leq) $P P'$
and *safe*: *safe_regex* m g r
and *mr*: *to_mregex* $r = (mr, \varphi s)$
and *pre_buf*: *wf_mbufn'* $\sigma P V j n R r buf$
and *pre_nts*: *list_all2* $(\lambda i t. t = \tau \sigma i)$ [*progress_regex* $\sigma P r j..<j'$] nts
and *xss*: *list_all3 list_all2*
 (*map* $(\lambda\varphi i. \text{qtable } n (fv \varphi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) i \varphi)) \varphi s)$
 (*map2 upt* (*map* $(\lambda\varphi. \text{progress } \sigma P \varphi j)$ $\varphi s)$ (*map* $(\lambda\varphi. \text{progress } \sigma P' \varphi j')$ $\varphi s)$) xss
and $j \leq j'$
shows *wf_mbufn'* $\sigma P' V j' n R r buf'$
and *wf_ts_regex* $\sigma P' j' r nts'$
 ⟨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 *bounded*: *pred_mapping* $(\lambda x. x \leq j)$ P *pred_mapping* $(\lambda x. x \leq j')$ P'
and *rm*: *rel_mapping* (\leq) $P P'$
and *pre_buf*: *wf_mbuf2'* $\sigma P V j n R \varphi \psi buf$
and *pre_nts*: *list_all2* $(\lambda i t. t = \tau \sigma i)$ [*min* (*progress* $\sigma P \varphi j$) (*progress* $\sigma P \psi j$).. $<j'$] nts
and *xs*: *list_all2* $(\lambda i. \text{qtable } n (\text{Formula.fv } \varphi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) i \varphi))$
 [*progress* $\sigma P \varphi j..<\text{progress } \sigma P' \varphi j'$] xs
and *ys*: *list_all2* $(\lambda i. \text{qtable } n (\text{Formula.fv } \psi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) i \psi))$
 [*progress* $\sigma P \psi j..<\text{progress } \sigma P' \psi j'$] ys
and $j \leq j'$
and *base*: $U (\text{min } (\text{progress } \sigma P \varphi j) (\text{progress } \sigma P \psi j)) z$
and *step*: $\bigwedge k X Y z. \text{min } (\text{progress } \sigma P \varphi j) (\text{progress } \sigma P \psi j) \leq k \implies$
 $Suc k \leq \text{progress } \sigma P' \varphi j' \implies Suc k \leq \text{progress } \sigma P' \psi j' \implies$
 $\text{qtable } n (\text{Formula.fv } \varphi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) k \varphi) X \implies$
 $\text{qtable } n (\text{Formula.fv } \psi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) k \psi) Y \implies$
 $U k z \implies U (Suc k) (f X Y (\tau \sigma k) z)$
shows $U (\text{min } (\text{progress } \sigma P' \varphi j') (\text{progress } \sigma P' \psi j')) z'$
 ⟨proof⟩

lemma *mbufnt_take_add_induct'*[consumes 6, case_names base step]:

assumes *eq*: *mbufnt_take* f z (*mbufn_add* xss buf) $nts = (z', buf', nts')$
and *bounded*: *pred_mapping* $(\lambda x. x \leq j)$ P *pred_mapping* $(\lambda x. x \leq j')$ P'
and *rm*: *rel_mapping* (\leq) $P P'$
and *safe*: *safe_regex* m g r

and *mr*: *to_mregex* *r* = (*mr*, φ s)
and *pre_buf*: *wf_mbufn'* σ *P* *V* *j* *n* *R* *r* *buf*
and *pre_nts*: *list_all2* ($\lambda i t. t = \tau \sigma i$) [*progress_regex* σ *P* *r* *j*..*j'*] *nts*
and *xss*: *list_all3* *list_all2*
(*map* ($\lambda \varphi i. qtable\ n\ (fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi)$) φ s)
(*map2* *upt* (*map* ($\lambda \varphi. progress\ \sigma\ P\ \varphi\ j$) φ s) (*map* ($\lambda \varphi. progress\ \sigma\ P'\ \varphi\ j'$) φ s)) *xss*)
and *j* \leq *j'*
and *base*: *U* (*progress_regex* σ *P* *r* *j*) *z*
and *step*: $\bigwedge k\ Xs\ z. progress_regex\ \sigma\ P\ r\ j\ \leq\ k \implies Suc\ k\ \leq\ progress_regex\ \sigma\ P'\ r\ j' \implies$
list_all2 ($\lambda \varphi. qtable\ n\ (Formula.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ k\ \varphi)$) φ s
Xs \implies
U *k* *z* \implies *U* (*Suc* *k*) (*f* *Xs* ($\tau\ \sigma\ k$) *z*)
shows *U* (*progress_regex* σ *P'* *r* *j'*) *z'*
<*proof*>

lemma *progress_Until_le*: *progress* σ *P* (*Formula.Until* φ *I* ψ) *j* \leq *min* (*progress* σ *P* φ *j*) (*progress* σ *P* ψ *j*)
<*proof*>

lemma *progress_MatchF_le*: *progress* σ *P* (*Formula.MatchF* *I* *r*) *j* \leq *progress_regex* σ *P* *r* *j*
<*proof*>

lemma *list_all2_upt_Cons*: *P* *a* *x* $\implies list_all2\ P\ [Suc\ a..**b]** *xs* $\implies Suc\ a\ \leq\ b \implies$
list_all2 *P* [*a*..*b*] (*x* # *xs*)
<*proof*>$

lemma *list_all2_upt_append*: *list_all2* *P* [*a*..*b*] *xs* $\implies list_all2\ P\ [b..**c]** *ys* \implies
a \leq *b* $\implies b \leq c \implies list_all2\ P\ [a..**c]** (*xs* @ *ys*)
<*proof*>$$

lemma *list_all3_list_all2_conv*: *list_all3* *R* *xs* *xs* *ys* = *list_all2* ($\lambda x. R\ x\ x$) *xs* *ys*
<*proof*>

lemma *map_split_map*: *map_split* *f* (*map* *g* *xs*) = *map_split* (*f* *o* *g*) *xs*
<*proof*>

lemma *map_split_alt*: *map_split* *f* *xs* = (*map* (*fst* *o* *f*) *xs*, *map* (*snd* *o* *f*) *xs*)
<*proof*>

lemma *fv_formula_of_constraint*: *fv* (*formula_of_constraint* (*t1*, *p*, *c*, *t2*)) = *fv_trm* *t1* \cup *fv_trm* *t2*
<*proof*>

lemma (*in* *maux*) *wf_mformula_wf_set*: *wf_mformula* σ *j* *P* *V* *n* *R* $\varphi\ \varphi' \implies wf_set\ n\ (Formula.fv\ \varphi')$
<*proof*>

lemma *qtable_mmulti_join*:

assumes *pos*: *list_all3* ($\lambda A\ Qi\ X. qtable\ n\ A\ P\ Qi\ X \wedge wf_set\ n\ A$) *A_pos* *Q_pos* *L_pos*
and *neg*: *list_all3* ($\lambda A\ Qi\ X. qtable\ n\ A\ P\ Qi\ X \wedge wf_set\ n\ A$) *A_neg* *Q_neg* *L_neg*
and *C_eq*: *C* = \bigcup (*set* *A_pos*) **and** *L_eq*: *L* = *L_pos* @ *L_neg*
and *A_pos* \neq [] **and** *fv_subset*: \bigcup (*set* *A_neg*) \subseteq \bigcup (*set* *A_pos*)
and *restrict_pos*: $\bigwedge x. wf_tuple\ n\ C\ x \implies P\ x \implies list_all\ (\lambda A. P\ (restrict\ A\ x))\ A_pos$
and *restrict_neg*: $\bigwedge x. wf_tuple\ n\ C\ x \implies P\ x \implies list_all\ (\lambda A. P\ (restrict\ A\ x))\ A_neg$
and *Qs*: $\bigwedge x. wf_tuple\ n\ C\ x \implies P\ x \implies Q\ x \longleftrightarrow$
list_all2 ($\lambda A\ Qi. Qi\ (restrict\ A\ x)$) *A_pos* *Q_pos* \wedge
list_all2 ($\lambda A\ Qi. \neg\ Qi\ (restrict\ A\ x)$) *A_neg* *Q_neg*
shows *qtable* *n* *C* *P* *Q* (*mmulti_join* *n* *A_pos* *A_neg* *L*)
<*proof*>

lemma *nth_filter*: $i < \text{length } (\text{filter } P \text{ } xs) \implies$
 $(\bigwedge i'. i' < \text{length } xs \implies P (xs ! i') \implies Q (xs ! i')) \implies Q (\text{filter } P \text{ } xs ! i)$
 <proof>

lemma *nth_partition*: $i < \text{length } xs \implies$
 $(\bigwedge i'. i' < \text{length } (\text{filter } P \text{ } xs) \implies Q (\text{filter } P \text{ } xs ! i')) \implies$
 $(\bigwedge i'. i' < \text{length } (\text{filter } (\text{Not } \circ P) \text{ } xs) \implies Q (\text{filter } (\text{Not } \circ P) \text{ } xs ! i')) \implies Q (xs ! i)$
 <proof>

lemma *qtable_bin_join*:
assumes $qtable \ n \ A \ P \ Q1 \ X \ qtable \ n \ B \ P \ Q2 \ Y \ \neg \ b \implies B \subseteq A \ C = A \cup B$
 $\bigwedge x. \text{wf_tuple } n \ C \ x \implies P \ x \implies P (\text{restrict } A \ x) \wedge P (\text{restrict } B \ x)$
 $\bigwedge x. b \implies \text{wf_tuple } n \ C \ x \implies P \ x \implies Q \ x \longleftrightarrow Q1 (\text{restrict } A \ x) \wedge Q2 (\text{restrict } B \ x)$
 $\bigwedge x. \neg b \implies \text{wf_tuple } n \ C \ x \implies P \ x \implies Q \ x \longleftrightarrow Q1 (\text{restrict } A \ x) \wedge \neg Q2 (\text{restrict } B \ x)$
shows $qtable \ n \ C \ P \ Q (\text{bin_join } n \ A \ X \ b \ B \ Y)$
 <proof>

lemma *restrict_update*: $y \notin A \implies y < \text{length } x \implies \text{restrict } A \ (x[y:=z]) = \text{restrict } A \ x$
 <proof>

lemma *qtable_assign*:
assumes $qtable \ n \ A \ P \ Q \ X$
 $y < n \ \text{insert } y \ A = A' \ y \notin A$
 $\bigwedge x'. \text{wf_tuple } n \ A' \ x' \implies P \ x' \implies P (\text{restrict } A \ x')$
 $\bigwedge x. \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \implies Q' (x[y:=\text{Some } (f \ x)])$
 $\bigwedge x'. \text{wf_tuple } n \ A' \ x' \implies P \ x' \implies Q' \ x' \implies Q (\text{restrict } A \ x') \wedge x' ! y = \text{Some } (f (\text{restrict } A \ x'))$
shows $qtable \ n \ A' \ P \ Q' ((\lambda x. x[y:=\text{Some } (f \ x)]) \ ' X) (\text{is } qtable \ _ \ _ \ _ \ ?Y)$
 <proof>

lemma *sat_the_update*: $y \notin \text{fv } \varphi \implies \text{Formula.sat } \sigma \ V (\text{map the } (x[y:=z])) \ i \ \varphi = \text{Formula.sat } \sigma \ V (\text{map the } x) \ i \ \varphi$
 <proof>

lemma *progress_constraint*: $\text{progress } \sigma \ P (\text{formula_of_constraint } c) \ j = j$
 <proof>

lemma *qtable_filter*:
assumes $qtable \ n \ A \ P \ Q \ X$
 $\bigwedge x. \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \wedge R \ x \longleftrightarrow Q' \ x$
shows $qtable \ n \ A \ P \ Q' (\text{Set.filter } R \ X) (\text{is } qtable \ _ \ _ \ _ \ ?Y)$
 <proof>

lemma *eval_constraint_sat_eq*: $\text{wf_tuple } n \ A \ x \implies \text{fv_trm } t1 \subseteq A \implies \text{fv_trm } t2 \subseteq A \implies$
 $\forall i \in A. i < n \implies \text{eval_constraint } (t1, p, c, t2) \ x =$
 $\text{Formula.sat } \sigma \ V (\text{map the } x) \ i (\text{formula_of_constraint } (t1, p, c, t2))$
 <proof>

declare *progress_le_gen*[simp]

definition *wf_envs* $\sigma \ j \ P \ P' \ V \ db =$
 $(\text{dom } V = \text{dom } P \wedge$
 $\text{Mapping.keys } db = \text{dom } P \cup \{p. p \in \text{fst } \Gamma \ \sigma \ j\} \wedge$
 $\text{rel_mapping } (\leq) \ P \ P' \wedge$
 $\text{pred_mapping } (\lambda i. i \leq j) \ P \wedge$
 $\text{pred_mapping } (\lambda i. i \leq \text{Suc } j) \ P' \wedge$
 $(\forall p \in \text{Mapping.keys } db - \text{dom } P. \text{the } (\text{Mapping.lookup } db \ p) = [\{ts. (p, ts) \in \Gamma \ \sigma \ j\}]) \wedge$
 $(\forall p \in \text{dom } P. \text{list_all2 } (\lambda i \ X. X = \text{the } (V \ p) \ i) [\text{the } (P \ p)..<\text{the } (P' \ p)] (\text{the } (\text{Mapping.lookup } db \ p))))$

lift_definition *mk_db* :: (Formula.name × event_data list) set ⇒ Formula.database is
 $\lambda X p. (if\ p \in\ fst\ 'X\ then\ Some\ [\{ts.\ (p,\ ts) \in\ X\}] \ else\ None)$ (proof)

lemma *wf_envs_mk_db*: *wf_envs* $\sigma\ j\ Map.empty\ Map.empty\ Map.empty\ (mk_db\ (\Gamma\ \sigma\ j))$
 (proof)

lemma *wf_envs_update*:

assumes *wf_envs*: *wf_envs* $\sigma\ j\ P\ P'\ V\ db$
and *m_eq*: $m = Formula.nfv\ \varphi$
and *in_fv*: $\{0..m\} \subseteq fv\ \varphi$
and *xs*: *list_all2* $(\lambda i.\ qtable\ m\ (Formula.fv\ \varphi))\ (mem_restr\ UNIV)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi)$
 $[progress\ \sigma\ P\ \varphi\ j..<progress\ \sigma\ P'\ \varphi\ (Suc\ j)]\ xs$
shows *wf_envs* $\sigma\ j\ (P(p \mapsto progress\ \sigma\ P\ \varphi\ j))\ (P'(p \mapsto progress\ \sigma\ P'\ \varphi\ (Suc\ j)))$
 $(V(p \mapsto \lambda i.\ \{v.\ length\ v = m \wedge Formula.sat\ \sigma\ V\ v\ i\ \varphi\}))$
 $(Mapping.update\ p\ (map\ (image\ (map\ the))\ xs)\ db)$
 (proof)

lemma *wf_envs_P_simps[simp]*:

wf_envs $\sigma\ j\ P\ P'\ V\ db \implies pred_mapping\ (\lambda i.\ i \leq j)\ P$
wf_envs $\sigma\ j\ P\ P'\ V\ db \implies pred_mapping\ (\lambda i.\ i \leq Suc\ j)\ P'$
wf_envs $\sigma\ j\ P\ P'\ V\ db \implies rel_mapping\ (\leq)\ P\ P'$
 (proof)

lemma *wf_envs_progress_le[simp]*:

wf_envs $\sigma\ j\ P\ P'\ V\ db \implies progress\ \sigma\ P\ \varphi\ j \leq j$
wf_envs $\sigma\ j\ P\ P'\ V\ db \implies progress\ \sigma\ P'\ \varphi\ (Suc\ j) \leq Suc\ j$
 (proof)

lemma *wf_envs_progress_regex_le[simp]*:

wf_envs $\sigma\ j\ P\ P'\ V\ db \implies progress_regex\ \sigma\ P\ r\ j \leq j$
wf_envs $\sigma\ j\ P\ P'\ V\ db \implies progress_regex\ \sigma\ P'\ r\ (Suc\ j) \leq Suc\ j$
 (proof)

lemma *wf_envs_progress_mono[simp]*:

wf_envs $\sigma\ j\ P\ P'\ V\ db \implies a \leq b \implies progress\ \sigma\ P\ \varphi\ a \leq progress\ \sigma\ P'\ \varphi\ b$
 (proof)

lemma *qtable_wf_tuple_cong*: *qtable* $n\ A\ P\ Q\ X \implies A = B \implies (\bigwedge v.\ wf_tuple\ n\ A\ v \implies P\ v \implies Q\ v = Q'\ v) \implies qtable\ n\ B\ P\ Q'\ X$
 (proof)

lemma (in *maux*) *meval*:

assumes *wf_mformula* $\sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi'\ wf_envs\ \sigma\ j\ P\ P'\ V\ db$
shows *case_meval* $n\ (\tau\ \sigma\ j)\ db\ \varphi\ of\ (xs,\ \varphi_n) \Rightarrow wf_mformula\ \sigma\ (Suc\ j)\ P'\ V\ n\ R\ \varphi_n\ \varphi' \wedge$
 $list_all2\ (\lambda i.\ qtable\ n\ (Formula.fv\ \varphi'))\ (mem_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi')$
 $[progress\ \sigma\ P\ \varphi'\ j..<progress\ \sigma\ P'\ \varphi'\ (Suc\ j)]\ xs$
 (proof)

6.6.4 Monitor step

lemma (in *maux*) *wf_mstate_mstep*: *wf_mstate* $\varphi\ \pi\ R\ st \implies last_ts\ \pi \leq snd\ tdb \implies$
wf_mstate $\varphi\ (psnoc\ \pi\ tdb)\ R\ (snd\ (mstep\ (map_prod\ mk_db\ id\ tdb)\ st))$
 (proof)

definition *flatten_verdicts* $Vs = (\bigcup\ (set\ (map\ (\lambda(i,\ X).\ (\lambda v.\ (i,\ v))\ 'X)\ Vs)))$

lemma *flatten_verdicts_append[simp]*:

$flatten_verdicts (Vs @ Us) = flatten_verdicts Vs \cup flatten_verdicts Us$
 ⟨proof⟩

lemma (in *maux*) *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 $(i, v) \in flatten_verdicts\ (fst\ (mstep\ (map_prod\ mk_db\ id\ tdb)\ st)) \longleftrightarrow$
 $progress\ \sigma\ Map.empty\ \varphi\ (plen\ \pi) \leq i \wedge i < progress\ \sigma\ Map.empty\ \varphi\ (Suc\ (plen\ \pi)) \wedge$
 $wf_tuple\ (Formula.nfv\ \varphi)\ (Formula.fv\ \varphi)\ v \wedge Formula.sat\ \sigma\ Map.empty\ (map\ the\ v)\ i\ \varphi$
 ⟨proof⟩

6.6.5 Monitor function

locale *verimon* = *verimon_spec* + *maux*

lemma (in *verimon*) *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 $(i, v) \in flatten_verdicts\ (fst\ (mstep\ (map_prod\ mk_db\ id\ tdb)\ st)) \longleftrightarrow$
 $(i, v) \in M\ (psnoc\ \pi\ tdb) - M\ \pi$
 ⟨proof⟩

context *maux*
begin

primrec *msteps0* **where**

msteps0 $[]\ st = ([], st)$
 | *msteps0* $(tdb\ \# \pi)\ st =$
 $(let\ (V', st') = mstep\ (map_prod\ mk_db\ id\ tdb)\ st;\ (V'', st'') = msteps0\ \pi\ st'\ in\ (V' @ V'', st''))$

primrec *msteps0_stateless* **where**

msteps0_stateless $[]\ st = []$
 | *msteps0_stateless* $(tdb\ \# \pi)\ st = (let\ (V', st') = mstep\ (map_prod\ mk_db\ id\ tdb)\ st\ in\ V' @ msteps0_stateless\ \pi\ st')$

lemma *msteps0_msteps0_stateless*: $fst\ (msteps0\ w\ st) = msteps0_stateless\ w\ st$
 ⟨proof⟩

lift_definition *msteps* :: $Formula.prefix \Rightarrow ('msaux, 'muaux)\ mstate \Rightarrow (nat \times event_data\ table)\ list \times$
 $('msaux, 'muaux)\ mstate$
is *msteps0* ⟨proof⟩

lift_definition *msteps_stateless* :: $Formula.prefix \Rightarrow ('msaux, 'muaux)\ mstate \Rightarrow (nat \times event_data\ table)\ list$
is *msteps0_stateless* ⟨proof⟩

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

lemma *msteps0_snoc*: *msteps0* $(\pi @ [tdb])\ st =$

$(let\ (V', st') = msteps0\ \pi\ st;\ (V'', st'') = mstep\ (map_prod\ mk_db\ id\ tdb)\ st'\ in\ (V' @ V'', st''))$
 ⟨proof⟩

lemma *msteps_psnoc*: *last_ts* $\pi \leq snd\ tdb \implies msteps\ (psnoc\ \pi\ tdb)\ st =$

$(let\ (V', st') = msteps\ \pi\ st;\ (V'', st'') = mstep\ (map_prod\ mk_db\ id\ tdb)\ st'\ in\ (V' @ V'', st''))$
 ⟨proof⟩

definition *monitor* **where**

monitor $\varphi\ \pi = msteps_stateless\ \pi\ (minit_safe\ \varphi)$

end

lemma *Suc_length_conv_snoc*: $(\text{Suc } n = \text{length } xs) = (\exists y \text{ ys. } xs = \text{ys} @ [y] \wedge \text{length } \text{ys} = n)$
<proof>

lemma (in *verimon*) *wf_mstate_msteps*: $\text{wf_mstate } \varphi \pi R st \implies \text{mem_restr } R v \implies \pi \leq \pi' \implies$
 $X = \text{msteps } (\text{pdrop } (\text{plen } \pi) \pi') st \implies \text{wf_mstate } \varphi \pi' R (\text{snd } X) \wedge$
 $((i, v) \in \text{flatten_verdicts } (\text{fst } X)) = ((i, v) \in M \pi' - M \pi)$
<proof>

lemma (in *verimon*) *wf_mstate_msteps_stateless*:
assumes $\text{wf_mstate } \varphi \pi R st \text{ mem_restr } R v \pi \leq \pi'$
shows $(i, v) \in \text{flatten_verdicts } (\text{msteps_stateless } (\text{pdrop } (\text{plen } \pi) \pi') st) \iff (i, v) \in M \pi' - M \pi$
<proof>

lemma (in *verimon*) *wf_mstate_msteps_stateless_UNIV*: $\text{wf_mstate } \varphi \pi \text{ UNIV } st \implies \pi \leq \pi' \implies$
 $\text{flatten_verdicts } (\text{msteps_stateless } (\text{pdrop } (\text{plen } \pi) \pi') st) = M \pi' - M \pi$
<proof>

lemma (in *verimon*) *mverdicts_Nil*: $M \text{ pnil} = \{\}$
<proof>

context *maux*
begin

lemma *minit_safe_minit*: $\text{mmonitorable } \varphi \implies \text{minit_safe } \varphi = \text{minit } \varphi$
<proof>

lemma *wf_mstate_minit_safe*: $\text{mmonitorable } \varphi \implies \text{wf_mstate } \varphi \text{ pnil } R (\text{minit_safe } \varphi)$
<proof>

end

lemma (in *verimon*) *monitor_mverdicts*: $\text{flatten_verdicts } (\text{monitor } \varphi \pi) = M \pi$
<proof>

6.7 Collected correctness results

context *verimon*
begin

We summarize the main results proved above.

1. The term M describes semantically the monitor's expected behaviour:
 - *mono_monitor*: $\pi \leq \pi' \implies M \pi \subseteq M \pi'$
 - *sound_monitor*: $\llbracket (i, v) \in M \pi; \text{prefix_of } \pi \sigma \rrbracket \implies \text{Formula.sat } \sigma (\lambda x. \text{None}) (\text{map the } v) i \varphi$
 - *complete_monitor*: $\llbracket \text{prefix_of } \pi \sigma; \text{wf_tuple } (\text{Formula.nfv } \varphi) (\text{fv } \varphi) v; \bigwedge \sigma. \text{prefix_of } \pi \sigma \implies \text{Formula.sat } \sigma (\lambda x. \text{None}) (\text{map the } v) i \varphi \rrbracket \implies \exists \pi'. \text{prefix_of } \pi' \sigma \wedge (i, v) \in M \pi'$
 - *sliceable_M*: $\text{mem_restr } S v \implies ((i, v) \in M (\text{pmap_}\Gamma (\lambda D. D \cap \text{relevant_events } \varphi S) \pi)) = ((i, v) \in M \pi)$
2. The executable monitor's online interface *minit_safe* and *mstep* preserves the invariant *wf_mstate* and produces the the verdicts according to M :

- $wf_mstate_minit_safe: mmonitorable \varphi' \implies wf_mstate \varphi' pnil R (minit_safe \varphi')$
- $wf_mstate_mstep: \llbracket wf_mstate \varphi' \pi R st; last_ts \pi \leq snd\ tdb \rrbracket \implies wf_mstate \varphi' (psnoc \pi\ tdb) R (snd (mstep (map_prod\ mk_db\ id\ tdb) st))$
- $mstep_mverdicts: \llbracket wf_mstate \varphi \pi R st; last_ts \pi \leq snd\ tdb; mem_restr R v \rrbracket \implies ((i, v) \in flatten_verdicts (fst (mstep (map_prod\ mk_db\ id\ tdb) st))) = ((i, v) \in M (psnoc \pi\ tdb) - M \pi)$

3. The executable monitor's offline interface *local.monitor* implements *M*:

- $monitor_mverdicts: flatten_verdicts (local.monitor \varphi \pi) = M \pi$

end

7 Efficient implementation of temporal operators

7.1 Optimized queue data structure

lemma *less_enat_iff*: $a < enat\ i \iff (\exists j. a = enat\ j \wedge j < i)$
 <proof>

type_synonym 'a queue_t = 'a list × 'a list

definition *queue_invariant* :: 'a queue_t ⇒ bool **where**
queue_invariant q = (case q of ([], []) ⇒ True | (fs, l # ls) ⇒ True | _ ⇒ False)

typedef 'a queue = {q :: 'a queue_t. queue_invariant q}
 <proof>

setup_lifting type_definition_queue

lift_definition *linearize* :: 'a queue ⇒ 'a list is $(\lambda q. case\ q\ of\ (fs, ls) \Rightarrow fs @ rev\ ls)$ <proof>

lift_definition *empty_queue* :: 'a queue is ([], [])
 <proof>

lemma *empty_queue_rep*: *linearize empty_queue* = []
 <proof>

lift_definition *is_empty* :: 'a queue ⇒ bool is $\lambda q. (case\ q\ of\ ([], []) \Rightarrow True \mid _ \Rightarrow False)$ <proof>

lemma *linearize_t_Nil*: $(case\ q\ of\ (fs, ls) \Rightarrow fs @ rev\ ls) = [] \iff q = ([], [])$
 <proof>

lemma *is_empty_alt*: $is_empty\ q \iff linearize\ q = []$
 <proof>

fun *prepend_queue_t* :: 'a ⇒ 'a queue_t ⇒ 'a queue_t **where**
prepend_queue_t a ([], []) = ([], [a])
 | *prepend_queue_t* a (fs, l # ls) = (a # fs, l # ls)
 | *prepend_queue_t* a (f # fs, []) = undefined

lift_definition *prepend_queue* :: 'a ⇒ 'a queue ⇒ 'a queue is *prepend_queue_t*
 <proof>

lemma *prepend_queue_rep*: *linearize (prepend_queue a q)* = a # *linearize q*
 <proof>

lift_definition *append_queue* :: 'a ⇒ 'a queue ⇒ 'a queue is
 (λa q. case q of (fs, ls) ⇒ (fs, a # ls))
 ⟨proof⟩

lemma *append_queue_rep*: linearize (append_queue a q) = linearize q @ [a]
 ⟨proof⟩

fun *safe_last_t* :: 'a queue_t ⇒ 'a option × 'a queue_t **where**
 safe_last_t ([], []) = (None, ([], []))
 | safe_last_t (fs, l # ls) = (Some l, (fs, l # ls))
 | safe_last_t (f # fs, []) = undefined

lift_definition *safe_last* :: 'a queue ⇒ 'a option × 'a queue is *safe_last_t*
 ⟨proof⟩

lemma *safe_last_rep*: safe_last q = (α, q') ⇒ linearize q = linearize q' ∧
 (case α of None ⇒ linearize q = [] | Some a ⇒ linearize q ≠ [] ∧ a = last (linearize q))
 ⟨proof⟩

fun *safe_hd_t* :: 'a queue_t ⇒ 'a option × 'a queue_t **where**
 safe_hd_t ([], []) = (None, ([], []))
 | safe_hd_t ([], [l]) = (Some l, ([], [l]))
 | safe_hd_t ([], l # ls) = (let fs = rev ls in (Some (hd fs), (fs, [l])))
 | safe_hd_t (f # fs, l # ls) = (Some f, (f # fs, l # ls))
 | safe_hd_t (f # fs, []) = undefined

lift_definition(code_dt) *safe_hd* :: 'a queue ⇒ 'a option × 'a queue is *safe_hd_t*
 ⟨proof⟩

lemma *safe_hd_rep*: safe_hd q = (α, q') ⇒ linearize q = linearize q' ∧
 (case α of None ⇒ linearize q = [] | Some a ⇒ linearize q ≠ [] ∧ a = hd (linearize q))
 ⟨proof⟩

fun *replace_hd_t* :: 'a ⇒ 'a queue_t ⇒ 'a queue_t **where**
 replace_hd_t a ([], []) = ([], [])
 | replace_hd_t a ([], [l]) = ([], [a])
 | replace_hd_t a ([], l # ls) = (let fs = rev ls in (a # tl fs, [l]))
 | replace_hd_t a (f # fs, l # ls) = (a # fs, l # ls)
 | replace_hd_t a (f # fs, []) = undefined

lift_definition *replace_hd* :: 'a ⇒ 'a queue ⇒ 'a queue is *replace_hd_t*
 ⟨proof⟩

lemma *tl_append*: xs ≠ [] ⇒ tl xs @ ys = tl (xs @ ys)
 ⟨proof⟩

lemma *replace_hd_rep*: linearize q = f # fs ⇒ linearize (replace_hd a q) = a # fs
 ⟨proof⟩

fun *replace_last_t* :: 'a ⇒ 'a queue_t ⇒ 'a queue_t **where**
 replace_last_t a ([], []) = ([], [])
 | replace_last_t a (fs, l # ls) = (fs, a # ls)
 | replace_last_t a (fs, []) = undefined

lift_definition *replace_last* :: 'a ⇒ 'a queue ⇒ 'a queue is *replace_last_t*
 ⟨proof⟩

lemma *replace_last_rep*: $\text{linearize } q = \text{fs} @ [f] \implies \text{linearize } (\text{replace_last } a \ q) = \text{fs} @ [a]$
 <proof>

fun *tl_queue_t* :: 'a queue_t \Rightarrow 'a queue_t **where**
tl_queue_t ([], []) = ([], [])
 | *tl_queue_t* ([], [l]) = ([], [l])
 | *tl_queue_t* ([], l # ls) = (tl (rev ls), [l])
 | *tl_queue_t* (a # as, fs) = (as, fs)

lift_definition *tl_queue* :: 'a queue \Rightarrow 'a queue **is** *tl_queue_t*
 <proof>

lemma *tl_queue_rep*: $\neg \text{is_empty } q \implies \text{linearize } (\text{tl_queue } q) = \text{tl } (\text{linearize } q)$
 <proof>

lemma *length_tl_queue_rep*: $\neg \text{is_empty } q \implies$
 $\text{length } (\text{linearize } (\text{tl_queue } q)) < \text{length } (\text{linearize } q)$
 <proof>

lemma *length_tl_queue_safe_hd*:
assumes *safe_hd* $q = (\text{Some } a, q')$
shows $\text{length } (\text{linearize } (\text{tl_queue } q')) < \text{length } (\text{linearize } q)$
 <proof>

function *dropWhile_queue* :: ('a \Rightarrow bool) \Rightarrow 'a queue \Rightarrow 'a queue **where**
dropWhile_queue $f \ q = (\text{case } \text{safe_hd } q \text{ of } (\text{None}, q') \Rightarrow q'$
 | (Some a, q') \Rightarrow if $f \ a$ then *dropWhile_queue* $f \ (\text{tl_queue } q')$ else $q')$
 <proof>

termination
 <proof>

lemma *dropWhile_hd_tl*: $xs \neq [] \implies$
 $\text{dropWhile } P \ xs = (\text{if } P \ (\text{hd } xs) \text{ then } \text{dropWhile } P \ (\text{tl } xs) \text{ else } xs)$
 <proof>

lemma *dropWhile_queue_rep*: $\text{linearize } (\text{dropWhile_queue } f \ q) = \text{dropWhile } f \ (\text{linearize } q)$
 <proof>

function *takeWhile_queue* :: ('a \Rightarrow bool) \Rightarrow 'a queue \Rightarrow 'a queue **where**
takeWhile_queue $f \ q = (\text{case } \text{safe_hd } q \text{ of } (\text{None}, q') \Rightarrow q'$
 | (Some a, q') \Rightarrow if $f \ a$
 then *prepend_queue* $a \ (\text{takeWhile_queue } f \ (\text{tl_queue } q'))$
 else *empty_queue*)
 <proof>

termination
 <proof>

lemma *takeWhile_hd_tl*: $xs \neq [] \implies$
 $\text{takeWhile } P \ xs = (\text{if } P \ (\text{hd } xs) \text{ then } \text{hd } xs \ \# \ \text{takeWhile } P \ (\text{tl } xs) \text{ else } [])$
 <proof>

lemma *takeWhile_queue_rep*: $\text{linearize } (\text{takeWhile_queue } f \ q) = \text{takeWhile } f \ (\text{linearize } q)$
 <proof>

function *takedropWhile_queue* :: ('a \Rightarrow bool) \Rightarrow 'a queue \Rightarrow 'a queue \times 'a list **where**
takedropWhile_queue $f \ q = (\text{case } \text{safe_hd } q \text{ of } (\text{None}, q') \Rightarrow (q', [])$
 | (Some a, q') \Rightarrow if $f \ a$
 then (case *takedropWhile_queue* $f \ (\text{tl_queue } q')$ of (q'', as) \Rightarrow (q'', a # as))

$else (q', [])$
 $\langle proof \rangle$
termination
 $\langle proof \rangle$

lemma *takedownWhile_queue_fst*: $fst (takedownWhile_queue f q) = dropWhile_queue f q$
 $\langle proof \rangle$

lemma *takedownWhile_queue_snd*: $snd (takedownWhile_queue f q) = takeWhile f (linearize q)$
 $\langle proof \rangle$

7.2 Optimized data structure for Since

type_synonym *'a mmsaux* = $ts \times ts \times bool\ list \times bool\ list \times$
 $(ts \times 'a\ table)\ queue \times (ts \times 'a\ table)\ queue \times$
 $(('a\ tuple, ts)\ mapping) \times (('a\ tuple, ts)\ mapping)$

fun *time_mmsaux* :: $'a\ mmsaux \Rightarrow ts$ **where**
 $time_mmsaux\ aux = (case\ aux\ of\ (nt, _) \Rightarrow nt)$

definition *ts_tuple_rel* :: $(ts \times 'a\ table)\ set \Rightarrow (ts \times 'a\ tuple)\ set$ **where**
 $ts_tuple_rel\ ys = \{(t, as). \exists X. as \in X \wedge (t, X) \in ys\}$

lemma *finite_fst_ts_tuple_rel*: $finite (fst \{tas \in ts_tuple_rel (set\ xs). P\ tas\})$
 $\langle proof \rangle$

lemma *ts_tuple_rel_ext_Cons*: $tas \in ts_tuple_rel \{(nt, X)\} \Longrightarrow$
 $tas \in ts_tuple_rel (set ((nt, X) \# tass))$
 $\langle proof \rangle$

lemma *ts_tuple_rel_ext_Cons'*: $tas \in ts_tuple_rel (set\ tass) \Longrightarrow$
 $tas \in ts_tuple_rel (set ((nt, X) \# tass))$
 $\langle proof \rangle$

lemma *ts_tuple_rel_intro*: $as \in X \Longrightarrow (t, X) \in ys \Longrightarrow (t, as) \in ts_tuple_rel\ ys$
 $\langle proof \rangle$

lemma *ts_tuple_rel_dest*: $(t, as) \in ts_tuple_rel\ ys \Longrightarrow \exists X. (t, X) \in ys \wedge as \in X$
 $\langle proof \rangle$

lemma *ts_tuple_rel_Un*: $ts_tuple_rel (ys \cup zs) = ts_tuple_rel\ ys \cup ts_tuple_rel\ zs$
 $\langle proof \rangle$

lemma *ts_tuple_rel_ext*: $tas \in ts_tuple_rel \{(nt, X)\} \Longrightarrow$
 $tas \in ts_tuple_rel (set ((nt, Y \cup X) \# tass))$
 $\langle proof \rangle$

lemma *ts_tuple_rel_ext'*: $tas \in ts_tuple_rel (set ((nt, X) \# tass)) \Longrightarrow$
 $tas \in ts_tuple_rel (set ((nt, X \cup Y) \# tass))$
 $\langle proof \rangle$

lemma *ts_tuple_rel_mono*: $ys \subseteq zs \Longrightarrow ts_tuple_rel\ ys \subseteq ts_tuple_rel\ zs$
 $\langle proof \rangle$

lemma *ts_tuple_rel_filter*: $ts_tuple_rel (set (filter (\lambda(t, X). P\ t) xs)) =$
 $\{(t, X) \in ts_tuple_rel (set\ xs). P\ t\}$
 $\langle proof \rangle$

lemma *ts_tuple_rel_set_filter*: $x \in ts_tuple_rel (set (filter P xs)) \implies x \in ts_tuple_rel (set xs)$
 <proof>

definition *valid_tuple* :: $(('a tuple, ts) mapping) \Rightarrow (ts \times 'a tuple) \Rightarrow bool$ **where**
valid_tuple tuple_since = $(\lambda(t, as). case Mapping.lookup tuple_since as of None \Rightarrow False | Some t' \Rightarrow t \geq t')$

definition *safe_max* :: $'a :: linorder set \Rightarrow 'a option$ **where**
safe_max X = $(if X = \{\} then None else Some (Max X))$

lemma *safe_max_empty*: $safe_max X = None \iff X = \{\}$
 <proof>

lemma *safe_max_empty_dest*: $safe_max X = None \implies X = \{\}$
 <proof>

lemma *safe_max_Some_intro*: $x \in X \implies \exists y. safe_max X = Some y$
 <proof>

lemma *safe_max_Some_dest_in*: $finite X \implies safe_max X = Some x \implies x \in X$
 <proof>

lemma *safe_max_Some_dest_le*: $finite X \implies safe_max X = Some x \implies y \in X \implies y \leq x$
 <proof>

fun *valid_mmsaux* :: $args \Rightarrow ts \Rightarrow 'a mmsaux \Rightarrow 'a Monitor.msaux \Rightarrow bool$ **where**
valid_mmsaux args cur (nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) ys \iff
 $(args_L args) \subseteq (args_R args) \wedge$
 $maskL = join_mask (args_n args) (args_L args) \wedge$
 $maskR = join_mask (args_n args) (args_R args) \wedge$
 $(\forall (t, X) \in set ys. table (args_n args) (args_R args) X) \wedge$
 $table (args_n args) (args_R args) (Mapping.keys tuple_in) \wedge$
 $table (args_n args) (args_R args) (Mapping.keys tuple_since) \wedge$
 $(\forall as \in \bigcup (snd ' (set (linearize data_prev))). wf_tuple (args_n args) (args_R args) as) \wedge$
 $cur = nt \wedge$
 $ts_tuple_rel (set ys) =$
 $\{tas \in ts_tuple_rel (set (linearize data_prev) \cup set (linearize data_in)).$
 $valid_tuple tuple_since tas\} \wedge$
 $sorted (map fst (linearize data_prev)) \wedge$
 $(\forall t \in fst ' set (linearize data_prev). t \leq nt \wedge nt - t < left (args_ivl args)) \wedge$
 $sorted (map fst (linearize data_in)) \wedge$
 $(\forall t \in fst ' set (linearize data_in). t \leq nt \wedge nt - t \geq left (args_ivl args)) \wedge$
 $(\forall as. Mapping.lookup tuple_in as = safe_max (fst ' \{tas \in ts_tuple_rel (set (linearize data_in)). valid_tuple tuple_since tas \wedge as = snd tas\})) \wedge$
 $(\forall as \in Mapping.keys tuple_since. case Mapping.lookup tuple_since as of Some t \Rightarrow t \leq nt)$

lemma *Mapping_lookup_filter_keys*: $k \in Mapping.keys (Mapping.filter f m) \implies Mapping.lookup (Mapping.filter f m) k = Mapping.lookup m k$
 <proof>

lemma *Mapping_filter_keys*: $(\forall k \in Mapping.keys m. P (Mapping.lookup m k)) \implies (\forall k \in Mapping.keys (Mapping.filter f m). P (Mapping.lookup (Mapping.filter f m) k))$
 <proof>

lemma *Mapping_filter_keys_le*: $(\bigwedge x. P x \implies P' x) \implies (\forall k \in Mapping.keys m. P (Mapping.lookup m k)) \implies (\forall k \in Mapping.keys m. P' (Mapping.lookup m k))$

<proof>

lemma *Mapping_keys_dest*: $x \in \text{Mapping.keys } f \implies \exists y. \text{Mapping.lookup } f \ x = \text{Some } y$
<proof>

lemma *Mapping_keys_intro*: $\text{Mapping.lookup } f \ x \neq \text{None} \implies x \in \text{Mapping.keys } f$
<proof>

lemma *valid_mmsaux_tuple_in_keys*: *valid_mmsaux* *args cur*
(*nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since*) *ys* \implies
Mapping.keys tuple_in = snd ' {tas \in ts_tuple_rel (set (linearize data_in)).
valid_tuple tuple_since tas}
<proof>

fun *init_mmsaux* :: *args* \Rightarrow 'a *mmsaux* **where**
init_mmsaux *args* = (0, 0, *join_mask* (*args_n* *args*) (*args_L* *args*),
join_mask (*args_n* *args*) (*args_R* *args*), *empty_queue*, *empty_queue*, *Mapping.empty*, *Mapping.empty*)

lemma *valid_init_mmsaux*: $L \subseteq R \implies \text{valid_mmsaux } (\text{init_args } I \ n \ L \ R \ b) \ 0$
(*init_mmsaux* (*init_args* *I n L R b*)) []
<proof>

abbreviation *filter_cond* $X' \ ts \ t' \equiv (\lambda as _. \neg (as \in X' \wedge \text{Mapping.lookup } ts \ as = \text{Some } t'))$

lemma *dropWhile_filter*:
sorted (*map fst xs*) $\implies \forall t \in \text{fst ' set } xs. t \leq nt \implies$
dropWhile ($\lambda(t, X). \text{enat } (nt - t) > c$) *xs* = *filter* ($\lambda(t, X). \text{enat } (nt - t) \leq c$) *xs*
<proof>

lemma *dropWhile_filter'*:
fixes *nt* :: *nat*
shows *sorted* (*map fst xs*) $\implies \forall t \in \text{fst ' set } xs. t \leq nt \implies$
dropWhile ($\lambda(t, X). nt - t \geq c$) *xs* = *filter* ($\lambda(t, X). nt - t < c$) *xs*
<proof>

lemma *dropWhile_filter''*:
sorted xs $\implies \forall t \in \text{set } xs. t \leq nt \implies$
dropWhile ($\lambda t. \text{enat } (nt - t) > c$) *xs* = *filter* ($\lambda t. \text{enat } (nt - t) \leq c$) *xs*
<proof>

lemma *takeWhile_filter*:
sorted (*map fst xs*) $\implies \forall t \in \text{fst ' set } xs. t \leq nt \implies$
takeWhile ($\lambda(t, X). \text{enat } (nt - t) > c$) *xs* = *filter* ($\lambda(t, X). \text{enat } (nt - t) > c$) *xs*
<proof>

lemma *takeWhile_filter'*:
fixes *nt* :: *nat*
shows *sorted* (*map fst xs*) $\implies \forall t \in \text{fst ' set } xs. t \leq nt \implies$
takeWhile ($\lambda(t, X). nt - t \geq c$) *xs* = *filter* ($\lambda(t, X). nt - t \geq c$) *xs*
<proof>

lemma *takeWhile_filter''*:
sorted xs $\implies \forall t \in \text{set } xs. t \leq nt \implies$
takeWhile ($\lambda t. \text{enat } (nt - t) > c$) *xs* = *filter* ($\lambda t. \text{enat } (nt - t) > c$) *xs*
<proof>

lemma *fold_Mapping_filter_None*: $\text{Mapping.lookup } ts \ as = \text{None} \implies$
Mapping.lookup (*fold* ($\lambda(t, X) \ ts. \text{Mapping.filter}$

$(\text{filter_cond } X \text{ } ts \text{ } t) \text{ } ts) \text{ } ds \text{ } ts) \text{ } as = \text{None}$
 <proof>

lemma *Mapping_lookup_filter_Some_P*: $\text{Mapping.lookup } (\text{Mapping.filter } P \text{ } m) \text{ } k = \text{Some } v \implies P \text{ } k \text{ } v$
 <proof>

lemma *Mapping_lookup_filter_None*: $(\bigwedge v. \neg P \text{ } k \text{ } v) \implies$
 $\text{Mapping.lookup } (\text{Mapping.filter } P \text{ } m) \text{ } k = \text{None}$
 <proof>

lemma *Mapping_lookup_filter_Some*: $(\bigwedge v. P \text{ } k \text{ } v) \implies$
 $\text{Mapping.lookup } (\text{Mapping.filter } P \text{ } m) \text{ } k = \text{Mapping.lookup } m \text{ } k$
 <proof>

lemma *Mapping_lookup_filter_not_None*: $\text{Mapping.lookup } (\text{Mapping.filter } P \text{ } m) \text{ } k \neq \text{None} \implies$
 $\text{Mapping.lookup } (\text{Mapping.filter } P \text{ } m) \text{ } k = \text{Mapping.lookup } m \text{ } k$
 <proof>

lemma *fold_Mapping_filter_Some_None*: $\text{Mapping.lookup } ts \text{ } as = \text{Some } t \implies$
 $as \in X \implies (t, X) \in \text{set } ds \implies$
 $\text{Mapping.lookup } (\text{fold } (\lambda(t, X) \text{ } ts. \text{Mapping.filter } (\text{filter_cond } X \text{ } ts \text{ } t) \text{ } ts) \text{ } ds \text{ } ts) \text{ } as = \text{None}$
 <proof>

lemma *fold_Mapping_filter_Some_Some*: $\text{Mapping.lookup } ts \text{ } as = \text{Some } t \implies$
 $(\bigwedge X. (t, X) \in \text{set } ds \implies as \notin X) \implies$
 $\text{Mapping.lookup } (\text{fold } (\lambda(t, X) \text{ } ts. \text{Mapping.filter } (\text{filter_cond } X \text{ } ts \text{ } t) \text{ } ts) \text{ } ds \text{ } ts) \text{ } as = \text{Some } t$
 <proof>

fun *shift_end* :: $args \Rightarrow ts \Rightarrow 'a \text{ mmsaux} \Rightarrow 'a \text{ mmsaux}$ **where**
 $\text{shift_end } args \text{ } nt \text{ } (t, gc, \text{maskL}, \text{maskR}, \text{data_prev}, \text{data_in}, \text{tuple_in}, \text{tuple_since}) =$
 (let $I = \text{args_ivl } args;$
 $\text{data_prev}' = \text{dropWhile_queue } (\lambda(t, X). \text{enat } (nt - t) > \text{right } I) \text{ } \text{data_prev};$
 $(\text{data_in}, \text{discard}) = \text{takedropWhile_queue } (\lambda(t, X). \text{enat } (nt - t) > \text{right } I) \text{ } \text{data_in};$
 $\text{tuple_in} = \text{fold } (\lambda(t, X) \text{ } \text{tuple_in}. \text{Mapping.filter}$
 $(\text{filter_cond } X \text{ } \text{tuple_in } t) \text{ } \text{tuple_in}) \text{ } \text{discard } \text{tuple_in } \text{in}$
 $(t, gc, \text{maskL}, \text{maskR}, \text{data_prev}', \text{data_in}, \text{tuple_in}, \text{tuple_since}))$

lemma *valid_shift_end_mmsaux_unfolded*:
assumes *valid_before*: $\text{valid_mmsaux } args \text{ } cur$
 $(ot, gc, \text{maskL}, \text{maskR}, \text{data_prev}, \text{data_in}, \text{tuple_in}, \text{tuple_since}) \text{ } \text{auxlist}$
and *nt_mono*: $nt \geq cur$
shows *valid_mmsaux* $args \text{ } cur \text{ } (\text{shift_end } args \text{ } nt$
 $(ot, gc, \text{maskL}, \text{maskR}, \text{data_prev}, \text{data_in}, \text{tuple_in}, \text{tuple_since}))$
 $(\text{filter } (\lambda(t, \text{rel}). \text{enat } (nt - t) \leq \text{right } (\text{args_ivl } args)) \text{ } \text{auxlist})$
 <proof>

lemma *valid_shift_end_mmsaux*: $\text{valid_mmsaux } args \text{ } cur \text{ } \text{aux } \text{auxlist} \implies nt \geq cur \implies$
 $\text{valid_mmsaux } args \text{ } cur \text{ } (\text{shift_end } args \text{ } nt \text{ } \text{aux})$
 $(\text{filter } (\lambda(t, \text{rel}). \text{enat } (nt - t) \leq \text{right } (\text{args_ivl } args)) \text{ } \text{auxlist})$
 <proof>

setup_lifting *type_definition_mapping*

lift_definition *upd_set* :: $('a, 'b) \text{ mapping} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow ('a, 'b) \text{ mapping}$ **is**
 $\lambda m \text{ } f \text{ } X \text{ } a. \text{if } a \in X \text{ then } \text{Some } (f \text{ } a) \text{ else } m \text{ } a$ <proof>

lemma *Mapping_lookup_upd_set*: $\text{Mapping.lookup } (\text{upd_set } m \text{ } f \text{ } X) \text{ } a =$
 (if $a \in X$ then $\text{Some } (f \text{ } a)$ else $\text{Mapping.lookup } m \text{ } a$)

<proof>

lemma *Mapping_upd_set_keys*: $Mapping.keys (upd_set\ m\ f\ X) = Mapping.keys\ m \cup X$
<proof>

lift_definition *upd_keys_on* :: $('a, 'b)$ *mapping* $\Rightarrow ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow$
 $('a, 'b)$ *mapping* **is**
 $\lambda m\ f\ X\ a.$ *case* *Mapping.lookup* *m* *a* *of* *Some* *b* \Rightarrow *Some* $(if\ a \in X\ then\ f\ a\ b\ else\ b)$
| *None* \Rightarrow *None* *<proof>*

lemma *Mapping_lookup_upd_keys_on*: $Mapping.lookup (upd_keys_on\ m\ f\ X)\ a =$
 $(case\ Mapping.lookup\ m\ a\ of\ Some\ b \Rightarrow Some\ (if\ a \in X\ then\ f\ a\ b\ else\ b) \mid None \Rightarrow None)$
<proof>

lemma *Mapping_upd_keys_sub*: $Mapping.keys (upd_keys_on\ m\ f\ X) = Mapping.keys\ m$
<proof>

lemma *fold_append_queue_rep*: $linearize (fold (\lambda x\ q.\ append_queue\ x\ q)\ xs\ q) = linearize\ q\ @\ xs$
<proof>

lemma *Max_Un_absorb*:
assumes *finite* *X* $X \neq \{\}$ *finite* *Y* $(\bigwedge x\ y.\ y \in Y \Longrightarrow x \in X \Longrightarrow y \leq x)$
shows $Max\ (X \cup Y) = Max\ X$
<proof>

lemma *Mapping_lookup_fold_upd_set_idle*: $\{(t, X) \in set\ xs.\ as \in Z\ X\ t\} = \{\} \Longrightarrow$
 $Mapping.lookup (fold (\lambda(t, X)\ m.\ upd_set\ m (\lambda_. t)\ (Z\ X\ t))\ xs\ m)\ as = Mapping.lookup\ m\ as$
<proof>

lemma *Mapping_lookup_fold_upd_set_max*: $\{(t, X) \in set\ xs.\ as \in Z\ X\ t\} \neq \{\} \Longrightarrow$
 $sorted (map\ fst\ xs) \Longrightarrow$
 $Mapping.lookup (fold (\lambda(t, X)\ m.\ upd_set\ m (\lambda_. t)\ (Z\ X\ t))\ xs\ m)\ as =$
 $Some (Max (fst ` $\{(t, X) \in set\ xs.\ as \in Z\ X\ t\}$))$
<proof>

fun *add_new_ts_mmsaux'* :: $args \Rightarrow ts \Rightarrow 'a\ mmsaux \Rightarrow 'a\ mmsaux$ **where**
 $add_new_ts_mmsaux'\ args\ nt (t, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) =$
 $(let\ I = args_ivl\ args;$
 $(data_prev, move) = takedownWhile_queue (\lambda(t, X).\ nt - t \geq left\ I)\ data_prev;$
 $data_in = fold (\lambda(t, X)\ data_in.\ append_queue (t, X)\ data_in)\ move\ data_in;$
 $tuple_in = fold (\lambda(t, X)\ tuple_in.\ upd_set\ tuple_in (\lambda_. t)$
 $\{as \in X.\ valid_tuple\ tuple_since (t, as)\})\ move\ tuple_in\ in$
 $(nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))$

lemma *Mapping_keys_fold_upd_set*: $k \in Mapping.keys (fold (\lambda(t, X)\ m.\ upd_set\ m (\lambda_. t)\ (Z\ t\ X))\ xs\ m) \Longrightarrow k \in Mapping.keys\ m \vee (\exists (t, X) \in set\ xs.\ k \in Z\ t\ X)$
<proof>

lemma *valid_add_new_ts_mmsaux'_unfolded*:
assumes *valid_before*: *valid_mmsaux* *args* *cur*
 $(ot, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since)$ *auxlist*
and *nt_mono*: $nt \geq cur$
shows *valid_mmsaux* *args* *nt* $(add_new_ts_mmsaux'\ args\ nt$
 $(ot, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))\ auxlist$
<proof>

lemma *valid_add_new_ts_mmsaux'*: *valid_mmsaux* *args* *cur* *aux* *auxlist* $\Longrightarrow nt \geq cur \Longrightarrow$
valid_mmsaux *args* *nt* $(add_new_ts_mmsaux'\ args\ nt\ aux)\ auxlist$

<proof>

definition *add_new_ts_mmsaux* :: *args* \Rightarrow *ts* \Rightarrow 'a *mmsaux* \Rightarrow 'a *mmsaux* **where**
add_new_ts_mmsaux *args* *nt* *aux* = *add_new_ts_mmsaux'* *args* *nt* (*shift_end* *args* *nt* *aux*)

lemma *valid_add_new_ts_mmsaux*:
assumes *valid_mmsaux* *args* *cur* *aux* *auxlist* *nt* \geq *cur*
shows *valid_mmsaux* *args* *nt* (*add_new_ts_mmsaux* *args* *nt* *aux*)
(*filter* ($\lambda(t, rel). \text{enat } (nt - t) \leq \text{right } (args_ivl \text{ args})$) *auxlist*)
<proof>

fun *join_mmsaux* :: *args* \Rightarrow 'a *table* \Rightarrow 'a *mmsaux* \Rightarrow 'a *mmsaux* **where**
join_mmsaux *args* *X* (*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
(*let* *pos* = *args_pos* *args* *in*
(*if* *maskL* = *maskR* *then*
(*let* *tuple_in* = *Mapping.filter* (*join_filter_cond* *pos* *X*) *tuple_in*;
tuple_since = *Mapping.filter* (*join_filter_cond* *pos* *X*) *tuple_since* *in*
(*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))
else if ($\forall i \in \text{set } maskL. \neg i$) *then*
(*let* *nones* = *replicate* (*length* *maskL*) *None*;
take_all = (*pos* \longleftrightarrow *nones* \in *X*);
tuple_in = (*if* *take_all* *then* *tuple_in* *else* *Mapping.empty*);
tuple_since = (*if* *take_all* *then* *tuple_since* *else* *Mapping.empty*) *in*
(*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))
else
(*let* *tuple_in* = *Mapping.filter* ($\lambda as _. \text{proj_tuple_in_join } pos \text{ maskL } as \text{ X}$) *tuple_in*;
tuple_since = *Mapping.filter* ($\lambda as _. \text{proj_tuple_in_join } pos \text{ maskL } as \text{ X}$) *tuple_since* *in*
(*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))))

fun *join_mmsaux_abs* :: *args* \Rightarrow 'a *table* \Rightarrow 'a *mmsaux* \Rightarrow 'a *mmsaux* **where**
join_mmsaux_abs *args* *X* (*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
(*let* *pos* = *args_pos* *args* *in*
(*let* *tuple_in* = *Mapping.filter* ($\lambda as _. \text{proj_tuple_in_join } pos \text{ maskL } as \text{ X}$) *tuple_in*;
tuple_since = *Mapping.filter* ($\lambda as _. \text{proj_tuple_in_join } pos \text{ maskL } as \text{ X}$) *tuple_since* *in*
(*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))

lemma *Mapping_filter_cong*:
assumes *cong*: ($\bigwedge k v. k \in \text{Mapping.keys } m \implies f k v = f' k v$)
shows *Mapping.filter* *f* *m* = *Mapping.filter* *f'* *m*
<proof>

lemma *join_mmsaux_abs_eq*:
assumes *valid_before*: *valid_mmsaux* *args* *cur*
(*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
and *table_left*: *table* (*args_n* *args*) (*args_L* *args*) *X*
shows *join_mmsaux* *args* *X* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
join_mmsaux_abs *args* *X* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*)
<proof>

lemma *valid_join_mmsaux_unfolded*:
assumes *valid_before*: *valid_mmsaux* *args* *cur*
(*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
and *table_left'*: *table* (*args_n* *args*) (*args_L* *args*) *X*
shows *valid_mmsaux* *args* *cur*
(*join_mmsaux* *args* *X* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))
(*map* ($\lambda(t, rel). (t, \text{join } rel \text{ (args_pos } args) \text{ X})$) *auxlist*)
<proof>

lemma *valid_join_mmsaux*: *valid_mmsaux* args cur aux auxlist \implies
table (args_n args) (args_L args) X \implies *valid_mmsaux* args cur
(join_mmsaux args X aux) (map ($\lambda(t, rel). (t, \text{join rel (args_pos args) X})$) auxlist)
<proof>

fun *gc_mmsaux* :: 'a mmsaux \Rightarrow 'a mmsaux **where**
gc_mmsaux (nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) =
(let all_tuples = \bigcup (snd ' (set (linearize data_prev) \cup set (linearize data_in)));
tuple_since' = Mapping.filter ($\lambda as _. as \in$ all_tuples) tuple_since in
(nt, nt, maskL, maskR, data_prev, data_in, tuple_in, tuple_since'))

lemma *valid_gc_mmsaux_unfolded*:
assumes *valid_before*: *valid_mmsaux* args cur (nt, gc, maskL, maskR, data_prev, data_in,
tuple_in, tuple_since) ys
shows *valid_mmsaux* args cur (*gc_mmsaux* (nt, gc, maskL, maskR, data_prev, data_in,
tuple_in, tuple_since)) ys
<proof>

lemma *valid_gc_mmsaux*: *valid_mmsaux* args cur aux ys \implies *valid_mmsaux* args cur (*gc_mmsaux* aux)
ys
<proof>

fun *gc_join_mmsaux* :: args \Rightarrow 'a table \Rightarrow 'a mmsaux \Rightarrow 'a mmsaux **where**
gc_join_mmsaux args X (t, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) =
(if enat (t - gc) > right (args_ivl args) then *join_mmsaux* args X (*gc_mmsaux* (t, gc, maskL, maskR,
data_prev, data_in, tuple_in, tuple_since))
else *join_mmsaux* args X (t, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))

lemma *gc_join_mmsaux_alt*: *gc_join_mmsaux* args rel1 aux = *join_mmsaux* args rel1 (*gc_mmsaux*
aux) \vee
gc_join_mmsaux args rel1 aux = *join_mmsaux* args rel1 aux
<proof>

lemma *valid_gc_join_mmsaux*:
assumes *valid_mmsaux* args cur aux auxlist *table* (args_n args) (args_L args) rel1
shows *valid_mmsaux* args cur (*gc_join_mmsaux* args rel1 aux)
(map ($\lambda(t, rel). (t, \text{join rel (args_pos args) rel1})$) auxlist)
<proof>

fun *add_new_table_mmsaux* :: args \Rightarrow 'a table \Rightarrow 'a mmsaux \Rightarrow 'a mmsaux **where**
add_new_table_mmsaux args X (t, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) =
(let tuple_since = upd_set tuple_since ($\lambda _. t$) (X - Mapping.keys tuple_since) in
(if 0 \geq left (args_ivl args) then (t, gc, maskL, maskR, data_prev, append_queue (t, X) data_in,
upd_set tuple_in ($\lambda _. t$) X, tuple_since)
else (t, gc, maskL, maskR, append_queue (t, X) data_prev, data_in, tuple_in, tuple_since)))

lemma *valid_add_new_table_mmsaux_unfolded*:
assumes *valid_before*: *valid_mmsaux* args cur
(nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) auxlist
and *table_X*: *table* (args_n args) (args_R args) X
shows *valid_mmsaux* args cur (*add_new_table_mmsaux* args X
(nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))
(case auxlist of
[] => [(cur, X)]
| ((t, y) # ts) => if t = cur then (t, y \cup X) # ts else (cur, X) # auxlist)
<proof>

lemma *valid_add_new_table_mmsaux*:

assumes *valid_before*: *valid_mmsaux* *args* *cur* *aux* *auxlist*
and *table_X*: *table* (*args_n* *args*) (*args_R* *args*) *X*
shows *valid_mmsaux* *args* *cur* (*add_new_table_mmsaux* *args* *X* *aux*)
 (case *auxlist* of
 [] => [(*cur*, *X*)]
 | ((*t*, *y*) # *ts*) => if *t* = *cur* then (*t*, *y* ∪ *X*) # *ts* else (*cur*, *X*) # *auxlist*)
 <proof>

lemma *foldr_ts_tuple_rel*:
as ∈ *foldr* (∪) (*concat* (*map* (λ(*t*, *rel*). if *P* *t* then [*rel*] else []) *auxlist*) {} ↔
 (∃ *t*. (*t*, *as*) ∈ *ts_tuple_rel* (*set* *auxlist*) ∧ *P* *t*)
 <proof>

lemma *image_snd*: (*a*, *b*) ∈ *X* ⇒ *b* ∈ *snd* ' *X*
 <proof>

fun *result_mmsaux* :: *args* ⇒ 'a *mmsaux* ⇒ 'a *table* **where**
result_mmsaux *args* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
Mapping.keys *tuple_in*

lemma *valid_result_mmsaux_unfolded*:
assumes *valid_mmsaux* *args* *cur*
 (*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
shows *result_mmsaux* *args* (*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
foldr (∪) [*rel*. (*t*, *rel*) ← *auxlist*, *left* (*args_ivl* *args*) ≤ *cur* - *t*] {}
 <proof>

lemma *valid_result_mmsaux*: *valid_mmsaux* *args* *cur* *aux* *auxlist* ⇒
result_mmsaux *args* *aux* = *foldr* (∪) [*rel*. (*t*, *rel*) ← *auxlist*, *left* (*args_ivl* *args*) ≤ *cur* - *t*] {}
 <proof>

interpretation *default_msaux*: *msaux* *valid_mmsaux* *init_mmsaux* *add_new_ts_mmsaux* *gc_join_mmsaux*
add_new_table_mmsaux *result_mmsaux*
 <proof>

7.3 Optimized data structure for Until

type_synonym *tp* = *nat*

type_synonym 'a *mmuaux* = *tp* × *ts* *queue* × *nat* × *bool* *list* × *bool* *list* ×
 ('a *tuple*, *tp*) *mapping* × (*tp*, ('a *tuple*, *ts* + *tp*) *mapping*) *mapping* × 'a *table* *list* × *nat*

definition *tstp_lt* :: *ts* + *tp* ⇒ *ts* ⇒ *tp* ⇒ *bool* **where**
tstp_lt *tstp* *ts* *tp* = *case_sum* (λ*ts'*. *ts'* ≤ *ts*) (λ*tp'*. *tp'* < *tp*) *tstp*

definition *tstp_le* :: *ts* + *tp* ⇒ *ts* ⇒ *tp* ⇒ *bool* **where**
tstp_le *tstp* *ts* *tp* = *case_sum* (λ*ts'*. *ts'* ≤ *ts*) (λ*tp'*. *tp'* ≤ *tp*) *tstp*

definition *ts_tp_lt* :: *ts* ⇒ *tp* ⇒ *ts* + *tp* ⇒ *bool* **where**
ts_tp_lt *ts* *tp* *tstp* = *case_sum* (λ*ts'*. *ts* ≤ *ts'*) (λ*tp'*. *tp* < *tp'*) *tstp*

definition *ts_tp_lt'* :: *ts* ⇒ *tp* ⇒ *ts* + *tp* ⇒ *bool* **where**
ts_tp_lt' *ts* *tp* *tstp* = *case_sum* (λ*ts'*. *ts* < *ts'*) (λ*tp'*. *tp* ≤ *tp'*) *tstp*

definition *ts_tp_le* :: *ts* ⇒ *tp* ⇒ *ts* + *tp* ⇒ *bool* **where**
ts_tp_le *ts* *tp* *tstp* = *case_sum* (λ*ts'*. *ts* ≤ *ts'*) (λ*tp'*. *tp* ≤ *tp'*) *tstp*

fun *max_tstp* :: *ts* + *tp* ⇒ *ts* + *tp* ⇒ *ts* + *tp* **where**

$max_tstp (Inl ts) (Inl ts') = Inl (max ts ts')$
 $| max_tstp (Inr tp) (Inr tp') = Inr (max tp tp')$
 $| max_tstp (Inl ts) _ = Inl ts$
 $| max_tstp _ (Inl ts) = Inl ts$

lemma max_tstp_idem : $max_tstp (max_tstp x y) y = max_tstp x y$
 $\langle proof \rangle$

lemma max_tstp_idem' : $max_tstp x (max_tstp x y) = max_tstp x y$
 $\langle proof \rangle$

lemma $max_tstp_d_d$: $max_tstp d d = d$
 $\langle proof \rangle$

lemma max_cases : $(max a b = a \implies P) \implies (max a b = b \implies P) \implies P$
 $\langle proof \rangle$

lemma max_tstpE : $isl tstp \longleftrightarrow isl tstp' \implies (max_tstp tstp tstp' = tstp \implies P) \implies$
 $(max_tstp tstp tstp' = tstp' \implies P) \implies P$
 $\langle proof \rangle$

lemma max_tstp_intro : $tstp_lt tstp ts tp \implies tstp_lt tstp' ts tp \implies isl tstp \longleftrightarrow isl tstp' \implies$
 $tstp_lt (max_tstp tstp tstp') ts tp$
 $\langle proof \rangle$

lemma max_tstp_intro' : $isl tstp \longleftrightarrow isl tstp' \implies$
 $ts_tp_le ts' tp' tstp \implies ts_tp_le ts' tp' (max_tstp tstp tstp')$
 $\langle proof \rangle$

lemma max_tstp_intro'' : $isl tstp \longleftrightarrow isl tstp' \implies$
 $ts_tp_le ts' tp' tstp' \implies ts_tp_le ts' tp' (max_tstp tstp tstp')$
 $\langle proof \rangle$

lemma max_tstp_intro''' : $isl tstp \longleftrightarrow isl tstp' \implies$
 $ts_tp_lt' ts' tp' tstp \implies ts_tp_lt' ts' tp' (max_tstp tstp tstp')$
 $\langle proof \rangle$

lemma max_tstp_intro'''' : $isl tstp \longleftrightarrow isl tstp' \implies$
 $ts_tp_lt' ts' tp' tstp' \implies ts_tp_lt' ts' tp' (max_tstp tstp tstp')$
 $\langle proof \rangle$

lemma max_tstp_isl : $isl tstp \longleftrightarrow isl tstp' \implies isl (max_tstp tstp tstp') \longleftrightarrow isl tstp$
 $\langle proof \rangle$

definition $filter_a1_map$:: $bool \Rightarrow tp \Rightarrow ('a\ tuple, tp) mapping \Rightarrow 'a\ table$ **where**
 $filter_a1_map\ pos\ tp\ a1_map =$
 $\{xs \in Mapping.keys\ a1_map.\ case\ Mapping.lookup\ a1_map\ xs\ of\ Some\ tp' \Rightarrow$
 $(pos \longrightarrow tp' \leq tp) \wedge (\neg pos \longrightarrow tp \leq tp')\}$

definition $filter_a2_map$:: $\mathcal{I} \Rightarrow ts \Rightarrow tp \Rightarrow (tp, ('a\ tuple, ts + tp) mapping) mapping \Rightarrow$
 $'a\ table$ **where**
 $filter_a2_map\ I\ ts\ tp\ a2_map = \{xs.\ \exists tp' \leq tp.\ (case\ Mapping.lookup\ a2_map\ tp'\ of\ Some\ m \Rightarrow$
 $(case\ Mapping.lookup\ m\ xs\ of\ Some\ tstp \Rightarrow ts_tp_lt'\ ts\ tp\ tstp\ | _ \Rightarrow False)$
 $| _ \Rightarrow False)\}$

fun $triple_eq_pair$:: $('a \times 'b \times 'c) \Rightarrow ('a \times 'd) \Rightarrow ('d \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'd \Rightarrow 'c) \Rightarrow bool$ **where**
 $triple_eq_pair\ (t, a1, a2)\ (ts', tp')\ f\ g \longleftrightarrow t = ts' \wedge a1 = f\ tp' \wedge a2 = g\ ts'\ tp'$

fun *valid_mmuauux'* :: *args* \Rightarrow *ts* \Rightarrow *ts* \Rightarrow 'a *mmuauux* \Rightarrow 'a *muauux* \Rightarrow *bool* **where**
valid_mmuauux' *args cur dt (tp, tss, len, maskL, maskR, a1_map, a2_map,*
done, done_length) auxlist \longleftrightarrow
args_L args \subseteq *args_R args* \wedge
maskL = *join_mask (args_n args) (args_L args)* \wedge
maskR = *join_mask (args_n args) (args_R args)* \wedge
len \leq *tp* \wedge
length (linearize tss) = *len* \wedge *sorted (linearize tss)* \wedge
 $(\forall t \in \text{set } (\text{linearize } tss). t \leq \text{cur} \wedge \text{enat } \text{cur} \leq \text{enat } t + \text{right } (\text{args_ivl } \text{args})) \wedge$
table (args_n args) (args_L args) (Mapping.keys a1_map) \wedge
Mapping.keys a2_map = $\{tp - \text{len}..tp\}$ \wedge
 $(\forall xs \in \text{Mapping.keys } a1_map. \text{case } \text{Mapping.lookup } a1_map \text{ } xs \text{ of } \text{Some } tp' \Rightarrow tp' < tp) \wedge$
 $(\forall tp' \in \text{Mapping.keys } a2_map. \text{case } \text{Mapping.lookup } a2_map \text{ } tp' \text{ of } \text{Some } m \Rightarrow$
table (args_n args) (args_R args) (Mapping.keys m) \wedge
 $(\forall xs \in \text{Mapping.keys } m. \text{case } \text{Mapping.lookup } m \text{ } xs \text{ of } \text{Some } tstp \Rightarrow$
tstp_lt tstp (cur - (left (args_ivl args) - 1)) tp \wedge $(\text{isl } tstp \longleftrightarrow \text{left } (\text{args_ivl } \text{args}) > 0)) \wedge$
length done = *done_length* \wedge *length done* + *len* = *length auxlist* \wedge
rev done = *map proj_thd (take (length done) auxlist)* \wedge
 $(\forall x \in \text{set } (\text{take } (\text{length } \text{done}) \text{ } \text{auxlist}). \text{check_before } (\text{args_ivl } \text{args}) \text{ } dt \text{ } x) \wedge$
sorted (map fst auxlist) \wedge
list_all2 $(\lambda x y. \text{triple_eq_pair } x \text{ } y \text{ } (\lambda tp'. \text{filter_a1_map } (\text{args_pos } \text{args}) \text{ } tp' \text{ } a1_map)$
 $(\lambda ts' tp'. \text{filter_a2_map } (\text{args_ivl } \text{args}) \text{ } ts' \text{ } tp' \text{ } a2_map)) (\text{drop } (\text{length } \text{done}) \text{ } \text{auxlist})$
 $(\text{zip } (\text{linearize } tss) [tp - \text{len}..<tp])$

definition *valid_muauux* :: *args* \Rightarrow *ts* \Rightarrow 'a *mmuauux* \Rightarrow 'a *muauux* \Rightarrow
bool **where**
valid_muauux args cur = *valid_mmuauux' args cur cur*

fun *eval_step_muauux* :: 'a *mmuauux* \Rightarrow 'a *muauux* **where**
eval_step_muauux (tp, tss, len, maskL, maskR, a1_map, a2_map,
done, done_length) = $(\text{case } \text{safe_hd } tss \text{ of } (\text{Some } ts, tss') \Rightarrow$
 $(\text{case } \text{Mapping.lookup } a2_map \text{ } (tp - \text{len}) \text{ of } \text{Some } m \Rightarrow$
let m = Mapping.filter $(\lambda _ tstp. ts_tp_lt' \text{ } ts \text{ } (tp - \text{len}) \text{ } tstp) \text{ } m;$
T = Mapping.keys m;
a2_map = Mapping.update $(tp - \text{len} + 1)$
 $(\text{case } \text{Mapping.lookup } a2_map \text{ } (tp - \text{len} + 1) \text{ of } \text{Some } m' \Rightarrow$
 $\text{Mapping.combine } (\lambda tstp \text{ } tstp'. \text{max_tstp } tstp \text{ } tstp') \text{ } m \text{ } m')$ *a2_map;*
a2_map = Mapping.delete $(tp - \text{len})$ *a2_map* *in*
 $(tp, \text{tl_queue } tss', \text{len} - 1, \text{maskL}, \text{maskR}, a1_map, a2_map,$
 $T \# \text{done}, \text{done_length} + 1))$

lemma *Mapping_update_keys*: *Mapping.keys (Mapping.update a b m)* = *Mapping.keys m* \cup $\{a\}$
<proof>

lemma *drop_is_Cons_take*: *drop n xs = y # ys* \implies *take (Suc n) xs = take n xs @ [y]*
<proof>

lemma *list_all2_weaken*: *list_all2 f xs ys* \implies
 $(\wedge x y. (x, y) \in \text{set } (\text{zip } xs \text{ } ys) \implies f \text{ } x \text{ } y \implies f' \text{ } x \text{ } y) \implies \text{list_all2 } f' \text{ } xs \text{ } ys$
<proof>

lemma *Mapping_lookup_delete*: *Mapping.lookup (Mapping.delete k m) k'* =
 $(\text{if } k = k' \text{ then } \text{None} \text{ else } \text{Mapping.lookup } m \text{ } k')$
<proof>

lemma *Mapping_lookup_update*: *Mapping.lookup (Mapping.update k v m) k'* =
 $(\text{if } k = k' \text{ then } \text{Some } v \text{ else } \text{Mapping.lookup } m \text{ } k')$
<proof>

lemma *hd_le_set*: $\text{sorted } xs \implies x \in \text{set } xs \implies \text{hd } xs \leq x$
 <proof>

lemma *Mapping_lookup_combineE*: $\text{Mapping.lookup } (\text{Mapping.combine } f \ m \ m') \ k = \text{Some } v \implies$
 $(\text{Mapping.lookup } m \ k = \text{Some } v \implies P) \implies$
 $(\text{Mapping.lookup } m' \ k = \text{Some } v \implies P) \implies$
 $(\bigwedge v' \ v''. \text{Mapping.lookup } m \ k = \text{Some } v' \implies \text{Mapping.lookup } m' \ k = \text{Some } v'' \implies$
 $f \ v' \ v'' = v \implies P) \implies P$
 <proof>

lemma *Mapping_keys_filterI*: $\text{Mapping.lookup } m \ k = \text{Some } v \implies f \ k \ v \implies$
 $k \in \text{Mapping.keys } (\text{Mapping.filter } f \ m)$
 <proof>

lemma *Mapping_keys_filterD*: $k \in \text{Mapping.keys } (\text{Mapping.filter } f \ m) \implies$
 $\exists v. \text{Mapping.lookup } m \ k = \text{Some } v \wedge f \ k \ v$
 <proof>

fun *lin_ts_mmuaux* :: 'a mmuaux \Rightarrow ts list **where**
lin_ts_mmuaux (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length) =
 linearize tss

lemma *valid_eval_step_mmuaux'*:
assumes *valid_mmuaux'* args cur dt aux auxlist
 $\text{lin_ts_mmuaux } aux = \text{ts} \# \text{tss}' \text{ enat } \text{ts} + \text{right } (\text{args_ivl } \text{args}) < dt$
shows *valid_mmuaux'* args cur dt (eval_step_mmuaux aux) auxlist \wedge
 $\text{lin_ts_mmuaux } (\text{eval_step_mmuaux } aux) = \text{tss}'$
 <proof>

lemma *done_empty_valid_mmuaux'_intro*:
assumes *valid_mmuaux'* args cur dt
 (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length) auxlist
shows *valid_mmuaux'* args cur dt'
 (tp, tss, len, maskL, maskR, a1_map, a2_map, [], 0)
 (drop (length done) auxlist)
 <proof>

lemma *valid_mmuaux'_mono*:
assumes *valid_mmuaux'* args cur dt aux auxlist $dt \leq dt'$
shows *valid_mmuaux'* args cur dt' aux auxlist
 <proof>

lemma *valid_foldl_eval_step_mmuaux'*:
assumes *valid_before*: *valid_mmuaux'* args cur dt aux auxlist
 $\text{lin_ts_mmuaux } aux = \text{tss} \ @ \ \text{tss}'$
 $\bigwedge \text{ts}. \text{ts} \in \text{set } (\text{take } (\text{length } \text{tss}) \ (\text{lin_ts_mmuaux } aux)) \implies \text{enat } \text{ts} + \text{right } (\text{args_ivl } \text{args}) < dt$
shows *valid_mmuaux'* args cur dt (foldl ($\lambda aux \ _. \ \text{eval_step_mmuaux } aux$) aux tss) auxlist \wedge
 $\text{lin_ts_mmuaux } (\text{foldl } (\lambda aux \ _. \ \text{eval_step_mmuaux } aux) aux \ \text{tss}) = \text{tss}'$
 <proof>

lemma *sorted_dropWhile_filter*: $\text{sorted } xs \implies \text{dropWhile } (\lambda t. \text{enat } t + \text{right } I < \text{enat } nt) \ xs =$
 $\text{filter } (\lambda t. \neg \text{enat } t + \text{right } I < \text{enat } nt) \ xs$
 <proof>

fun *shift_mmuaux* :: args \Rightarrow ts \Rightarrow 'a mmuaux \Rightarrow 'a mmuaux **where**
shift_mmuaux args nt (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length) =
 (let tss_list = linearize (takeWhile_queue ($\lambda t. \text{enat } t + \text{right } (\text{args_ivl } \text{args}) < \text{enat } nt$) tss) in

$foldl (\lambda aux _. eval_step_mmuaux\ aux) (tp, tss, len, maskL, maskR,$
 $a1_map, a2_map, done, done_length) tss_list)$

lemma *valid_shift_mmuaux'*:

assumes *valid_mmuaux' args cur cur aux auxlist nt \geq cur*

shows *valid_mmuaux' args cur nt (shift_mmuaux args nt aux) auxlist \wedge*

($\forall ts \in set (lin_ts_mmuaux (shift_mmuaux\ args\ nt\ aux)). \neg enat\ ts + right (args_ivl\ args) < nt$)

<proof>

lift_definition *upd_set'* :: *('a, 'b) mapping \Rightarrow 'b \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a set \Rightarrow ('a, 'b) mapping* **is**

*$\lambda m\ d\ f\ X\ a.$ (if $a \in X$ then (case *Mapping.lookup* *m a* of *Some b* \Rightarrow *Some (f b)* | *None* \Rightarrow *Some d*)*
*else *Mapping.lookup m a*) <proof>*

lemma *upd_set'_lookup*: *Mapping.lookup (upd_set' m d f X) a = (if $a \in X$ then*

*(case *Mapping.lookup m a* of *Some b* \Rightarrow *Some (f b)* | *None* \Rightarrow *Some d*) else *Mapping.lookup m a*)*

<proof>

lemma *upd_set'_keys*: *Mapping.keys (upd_set' m d f X) = Mapping.keys m \cup X*

<proof>

lift_definition *upd_nested* :: *('a, ('b, 'c) mapping) mapping \Rightarrow*

'c \Rightarrow ('c \Rightarrow 'c) \Rightarrow ('a \times 'b) set \Rightarrow ('a, ('b, 'c) mapping) mapping **is**

*$\lambda m\ d\ f\ X\ a.$ case *Mapping.lookup m a* of *Some m'* \Rightarrow *Some (upd_set' m' d f {b. (a, b) \in X})**

*| *None* \Rightarrow if $a \in fst\ 'X$ then *Some (upd_set' Mapping.empty d f {b. (a, b) \in X})* else *None* <proof>*

lemma *upd_nested_lookup*: *Mapping.lookup (upd_nested m d f X) a =*

*(case *Mapping.lookup m a* of *Some m'* \Rightarrow *Some (upd_set' m' d f {b. (a, b) \in X})*)*

*| *None* \Rightarrow if $a \in fst\ 'X$ then *Some (upd_set' Mapping.empty d f {b. (a, b) \in X})* else *None*)*

<proof>

lemma *upd_nested_keys*: *Mapping.keys (upd_nested m d f X) = Mapping.keys m \cup fst 'X*

<proof>

fun *add_new_mmuaux* :: *args \Rightarrow 'a table \Rightarrow 'a table \Rightarrow ts \Rightarrow 'a mmuaux \Rightarrow 'a mmuaux* **where**

add_new_mmuaux args rel1 rel2 nt aux =

(let (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length) =

shift_mmuaux args nt aux;

I = args_ivl args; pos = args_pos args;

new_tstp = (if left I = 0 then Inr tp else Inl (nt - (left I - 1)));

*tmp = \bigcup (($\lambda as.$ case *Mapping.lookup a1_map (proj_tuple maskL as)* of *None* \Rightarrow*

(if $\neg pos$ then {(tp - len, as)} else {}))

*| *Some tp'* \Rightarrow if pos then {(max (tp - len) tp', as)}*

else {(max (tp - len) (tp' + 1), as)} ' rel2) \cup (if left I = 0 then {tp} \times rel2 else {});

a2_map = Mapping.update (tp + 1) Mapping.empty

(upd_nested a2_map new_tstp (max_tstp new_tstp tmp);

a1_map = (if pos then Mapping.filter ($\lambda as _.$ as \in rel1)

(upd_set a1_map ($\lambda _.$ tp) (rel1 - Mapping.keys a1_map)) else upd_set a1_map ($\lambda _.$ tp) rel1);

tss = append_queue nt tss in

(tp + 1, tss, len + 1, maskL, maskR, a1_map, a2_map, done, done_length))

lemma *fst_case*: *($\lambda x.$ fst (case x of (t, a1, a2) \Rightarrow (t, y t a1 a2, z t a1 a2))) = fst*

<proof>

lemma *list_all2_in_setE*: *list_all2 P xs ys \Longrightarrow x \in set xs \Longrightarrow ($\bigwedge y.$ y \in set ys \Longrightarrow P x y \Longrightarrow Q) \Longrightarrow Q*

<proof>

lemma *list_all2_zip*: *list_all2 ($\lambda x y.$ triple_eq_pair x y f g) xs (zip ys zs) \Longrightarrow*

($\bigwedge y.$ y \in set ys \Longrightarrow Q y) \Longrightarrow x \in set xs \Longrightarrow Q (fst x)

<proof>

lemma *list_appendE*: $xs = ys @ zs \implies x \in \text{set } xs \implies$
 $(x \in \text{set } ys \implies P) \implies (x \in \text{set } zs \implies P) \implies P$
<proof>

lemma *take_takeWhile*: $n \leq \text{length } ys \implies$
 $(\bigwedge y. y \in \text{set } (\text{take } n \text{ } ys) \implies P \ y) \implies$
 $(\bigwedge y. y \in \text{set } (\text{drop } n \text{ } ys) \implies \neg P \ y) \implies$
 $\text{take } n \text{ } ys = \text{takeWhile } P \ ys$
<proof>

lemma *valid_add_new_mmuaux*:
assumes *valid_before*: *valid_mmuaux* *args* *cur* *aux* *auxlist*
and *tabs*: *table* (*args_n* *args*) (*args_L* *args*) *rel1* *table* (*args_n* *args*) (*args_R* *args*) *rel2*
and *nt_mono*: $nt \geq \text{cur}$
shows *valid_mmuaux* *args* *nt* (*add_new_mmuaux* *args* *rel1* *rel2* *nt* *aux*)
(*update_until* *args* *rel1* *rel2* *nt* *auxlist*)
<proof>

lemma *list_all2_check_before*: *list_all2* $(\lambda x y. \text{triple_eq_pair } x \ y \ f \ g) \ xs \ (\text{zip } ys \ zs) \implies$
 $(\bigwedge y. y \in \text{set } ys \implies \neg \text{enat } y + \text{right } I < nt) \implies x \in \text{set } xs \implies \neg \text{check_before } I \ nt \ x$
<proof>

fun *eval_mmuaux* :: *args* \Rightarrow *ts* \Rightarrow 'a *mmuaux* \Rightarrow 'a *table* *list* \times 'a *mmuaux* **where**
eval_mmuaux *args* *nt* *aux* =
(*let* (*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*, *done*, *done_length*) =
shift_mmuaux *args* *nt* *aux* *in*
(*rev* *done*, (*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*, [], 0)))

lemma *valid_eval_mmuaux*:
assumes *valid_mmuaux* *args* *cur* *aux* *auxlist* $nt \geq \text{cur}$
eval_mmuaux *args* *nt* *aux* = (*res*, *aux'*) *eval_until* (*args_ivl* *args*) *nt* *auxlist* = (*res'*, *auxlist'*)
shows *res* = *res'* \wedge *valid_mmuaux* *args* *cur* *aux'* *auxlist'*
<proof>

definition *init_mmuaux* :: *args* \Rightarrow 'a *mmuaux* **where**
init_mmuaux *args* = (0, *empty_queue*, 0,
join_mask (*args_n* *args*) (*args_L* *args*), *join_mask* (*args_n* *args*) (*args_R* *args*),
Mapping.empty, *Mapping.update* 0 *Mapping.empty* *Mapping.empty*, [], 0)

lemma *valid_init_mmuaux*: $L \subseteq R \implies \text{valid_mmuaux } (\text{init_args } I \ n \ L \ R \ b) \ 0$
(*init_mmuaux* (*init_args* *I* *n* *L* *R* *b*)) []
<proof>

fun *length_mmuaux* :: *args* \Rightarrow 'a *mmuaux* \Rightarrow *nat* **where**
length_mmuaux *args* (*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*, *done*, *done_length*) =
len + *done_length*

lemma *valid_length_mmuaux*:
assumes *valid_mmuaux* *args* *cur* *aux* *auxlist*
shows *length_mmuaux* *args* *aux* = *length* *auxlist*
<proof>

8 Instantiation of the generic algorithm and code setup

instantiation *enat* :: *set_impl* **begin**
definition *set_impl_enat* :: (*enat*, *set_impl*) *phantom* **where**

```

set_impl_enat = phantom set_RBT

instance <proof>
end

derive ccompare Formula.trm
derive (eq) ceq Formula.trm
derive (rbt) set_impl Formula.trm
derive (eq) ceq Monitor.mregex
derive ccompare Monitor.mregex
derive (rbt) set_impl Monitor.mregex
derive (rbt) mapping_impl Monitor.mregex
derive (no) cenum Monitor.mregex
derive (rbt) set_impl event_data
derive (rbt) mapping_impl event_data

definition add_new_mmuaux' :: args ⇒ event_data table ⇒ event_data table ⇒ ts ⇒ event_data
mmuaux ⇒
  event_data mmuaux where
  add_new_mmuaux' = add_new_mmuaux

interpretation muaux valid_mmuaux init_mmuaux add_new_mmuaux' length_mmuaux eval_mmuaux
<proof>

type_synonym 'a vmsaux = nat × (nat × 'a table) list

definition valid_vmsaux :: args ⇒ nat ⇒ event_data vmsaux ⇒
(nat × event_data table) list ⇒ bool where
  valid_vmsaux = (λ_ cur (t, aux) auxlist. t = cur ∧ aux = auxlist)

definition init_vmsaux :: args ⇒ event_data vmsaux where
  init_vmsaux = (λ_. (0, []))

definition add_new_ts_vmsaux :: args ⇒ nat ⇒ event_data vmsaux ⇒ event_data vmsaux where
  add_new_ts_vmsaux = (λargs nt (t, auxlist). (nt, filter (λ(t, rel).
  enat (nt - t) ≤ right (args_ivl args)) auxlist))

definition join_vmsaux :: args ⇒ event_data table ⇒ event_data vmsaux ⇒ event_data vmsaux where
  join_vmsaux = (λargs rel1 (t, auxlist). (t, map (λ(t, rel).
  (t, join rel (args_pos args) rel1)) auxlist))

definition add_new_table_vmsaux :: args ⇒ event_data table ⇒ event_data vmsaux ⇒
event_data vmsaux where
  add_new_table_vmsaux = (λargs rel2 (cur, auxlist). (cur, (case auxlist of
  [] => [(cur, rel2)]
  | ((t, y) # ts) => if t = cur then (t, y ∪ rel2) # ts else (cur, rel2) # auxlist)))

definition result_vmsaux :: args ⇒ event_data vmsaux ⇒ event_data table where
  result_vmsaux = (λargs (cur, auxlist).
  foldr (∪) [rel. (t, rel) ← auxlist, left (args_ivl args) ≤ cur - t] {})

type_synonym 'a vmuaux = nat × (nat × 'a table × 'a table) list

definition valid_vmuaux :: args ⇒ nat ⇒ event_data vmuaux ⇒
(nat × event_data table × event_data table) list ⇒ bool where
  valid_vmuaux = (λ_ cur (t, aux) auxlist. t = cur ∧ aux = auxlist)

definition init_vmuaux :: args ⇒ event_data vmuaux where

```

$init_vmuaux = (\lambda_ . (0, []))$

definition $add_new_vmuaux :: args \Rightarrow event_data\ table \Rightarrow event_data\ table \Rightarrow nat \Rightarrow event_data\ vmuaux \Rightarrow event_data\ vmuaux$ **where**
 $add_new_vmuaux = (\lambda args\ rel1\ rel2\ nt\ (t,\ auxlist). (nt,\ update_until\ args\ rel1\ rel2\ nt\ auxlist))$

definition $length_vmuaux :: args \Rightarrow event_data\ vmuaux \Rightarrow nat$ **where**
 $length_vmuaux = (\lambda_ (_,\ auxlist). length\ auxlist)$

definition $eval_vmuaux :: args \Rightarrow nat \Rightarrow event_data\ vmuaux \Rightarrow event_data\ table\ list \times event_data\ vmuaux$ **where**
 $eval_vmuaux = (\lambda args\ nt\ (t,\ auxlist). (let\ (res,\ auxlist') = eval_until\ (args_ivl\ args)\ nt\ auxlist\ in\ (res,\ (t,\ auxlist'))))$

global_interpretation $verimon_maux: mau\ valid_vmsaux\ init_vmsaux\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ result_vmsaux\ valid_vmuaux\ init_vmuaux\ add_new_vmuaux\ length_vmuaux\ eval_vmuaux$
defines $vminit0 = mau.minit0\ (init_vmsaux :: _ \Rightarrow event_data\ vmsaux)\ (init_vmuaux :: _ \Rightarrow event_data\ vmuaux) :: _ \Rightarrow Formula.formula \Rightarrow _$
and $vmin0 = mau.minit\ (init_vmsaux :: _ \Rightarrow event_data\ vmsaux)\ (init_vmuaux :: _ \Rightarrow event_data\ vmuaux) :: Formula.formula \Rightarrow _$
and $vmin0_safe = mau.minit_safe\ (init_vmsaux :: _ \Rightarrow event_data\ vmsaux)\ (init_vmuaux :: _ \Rightarrow event_data\ vmuaux) :: Formula.formula \Rightarrow _$
and $vmupdate_since = mau.update_since\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ (result_vmsaux :: _ \Rightarrow event_data\ vmsaux \Rightarrow event_data\ table)$
and $vmeval = mau.meval\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ (result_vmsaux :: _ \Rightarrow event_data\ vmsaux \Rightarrow _)\ add_new_vmuaux\ (eval_vmuaux :: _ \Rightarrow _ \Rightarrow event_data\ vmuaux \Rightarrow _)$
and $vmstep = mau.mstep\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ (result_vmsaux :: _ \Rightarrow event_data\ vmsaux \Rightarrow _)\ add_new_vmuaux\ (eval_vmuaux :: _ \Rightarrow _ \Rightarrow event_data\ vmuaux \Rightarrow _)$
and $vmsteps0_stateless = mau.msteps0_stateless\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ (result_vmsaux :: _ \Rightarrow event_data\ vmsaux \Rightarrow _)\ add_new_vmuaux\ (eval_vmuaux :: _ \Rightarrow _ \Rightarrow event_data\ vmuaux \Rightarrow _)$
and $vmsteps_stateless = mau.msteps_stateless\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ (result_vmsaux :: _ \Rightarrow event_data\ vmsaux \Rightarrow _)\ add_new_vmuaux\ (eval_vmuaux :: _ \Rightarrow _ \Rightarrow event_data\ vmuaux \Rightarrow _)$
and $vmonitor = mau.monitor\ init_vmsaux\ add_new_ts_vmsaux\ join_vmsaux\ add_new_table_vmsaux\ (result_vmsaux :: _ \Rightarrow event_data\ vmsaux \Rightarrow _)\ init_vmuaux\ add_new_vmuaux\ (eval_vmuaux :: _ \Rightarrow _ \Rightarrow event_data\ vmuaux \Rightarrow _)$
 $\langle proof \rangle$

global_interpretation $default_maux: mau\ valid_mmsaux\ init_mmsaux :: _ \Rightarrow event_data\ mmsaux\ add_new_ts_mmsaux\ gc_join_mmsaux\ add_new_table_mmsaux\ result_mmsaux\ valid_mmuaux\ init_mmuaux :: _ \Rightarrow event_data\ mmuaux\ add_new_mmuaux'\ length_mmuaux\ eval_mmuaux$
defines $minit0 = mau.minit0\ (init_mmsaux :: _ \Rightarrow event_data\ mmsaux)\ (init_mmuaux :: _ \Rightarrow event_data\ mmuaux) :: _ \Rightarrow Formula.formula \Rightarrow _$
and $minit = mau.minit\ (init_mmsaux :: _ \Rightarrow event_data\ mmsaux)\ (init_mmuaux :: _ \Rightarrow event_data\ mmuaux) :: Formula.formula \Rightarrow _$
and $minit_safe = mau.minit_safe\ (init_mmsaux :: _ \Rightarrow event_data\ mmsaux)\ (init_mmuaux :: _ \Rightarrow event_data\ mmuaux) :: Formula.formula \Rightarrow _$
and $mupdate_since = mau.update_since\ add_new_ts_mmsaux\ gc_join_mmsaux\ add_new_table_mmsaux\ (result_mmsaux :: _ \Rightarrow event_data\ mmsaux \Rightarrow event_data\ table)$
and $meval = mau.meval\ add_new_ts_mmsaux\ gc_join_mmsaux\ add_new_table_mmsaux\ (result_mmsaux :: _ \Rightarrow event_data\ mmsaux \Rightarrow _)\ add_new_mmuaux'\ (eval_mmuaux :: _ \Rightarrow _ \Rightarrow event_data\ mmuaux \Rightarrow _)$
and $mstep = mau.mstep\ add_new_ts_mmsaux\ gc_join_mmsaux\ add_new_table_mmsaux\ (result_mmsaux :: _ \Rightarrow event_data\ mmsaux \Rightarrow _)\ add_new_mmuaux'\ (eval_mmuaux :: _ \Rightarrow _ \Rightarrow event_data\ mmuaux \Rightarrow _)$

$\Rightarrow _)$
and $msteps0_stateless = maux.msteps0_stateless \text{ add_new_ts_mmsaux } gc_join_mmsaux \text{ add_new_table_mmsaux}$
 $(result_mmsaux :: _ \Rightarrow event_data \text{ mmsaux } \Rightarrow _) \text{ add_new_mmuaux}' (eval_mmuaux :: _ \Rightarrow _ \Rightarrow$
 $event_data \text{ mmuaux } \Rightarrow _)$
and $msteps_stateless = maux.msteps_stateless \text{ add_new_ts_mmsaux } gc_join_mmsaux \text{ add_new_table_mmsaux}$
 $(result_mmsaux :: _ \Rightarrow event_data \text{ mmsaux } \Rightarrow _) \text{ add_new_mmuaux}' (eval_mmuaux :: _ \Rightarrow _ \Rightarrow$
 $event_data \text{ mmuaux } \Rightarrow _)$
and $monitor = maux.monitor \text{ init_mmsaux } add_new_ts_mmsaux \text{ gc_join_mmsaux } add_new_table_mmsaux$
 $(result_mmsaux :: _ \Rightarrow event_data \text{ mmsaux } \Rightarrow _) \text{ init_mmuaux } add_new_mmuaux' (eval_mmuaux ::$
 $_ \Rightarrow _ \Rightarrow event_data \text{ mmuaux } \Rightarrow _)$
 $\langle proof \rangle$

lemma $image_these: f ' Option.these X = Option.these (map_option f ' X)$
 $\langle proof \rangle$

thm $default_maux.meval.simps(2)$

lemma $meval_MPred: meval \ n \ t \ db \ (MPred \ e \ ts) =$
 $(case \ Mapping.lookup \ db \ e \ of \ None \Rightarrow \{\} \mid \ Some \ Xs \Rightarrow \text{map} \ (\lambda X. \bigcup v \in X.$
 $\text{set_option} \ (\text{map_option} \ (\lambda f. \ Table.tabulate \ f \ 0 \ n) \ (\text{match} \ ts \ v)))) \ Xs, \ MPred \ e \ ts)$
 $\langle proof \rangle$

lemmas $meval_code[code] = default_maux.meval.simps(1) \ meval_MPred \ default_maux.meval.simps(3-)$

definition $mk_db :: (Formula.name \times event_data \ list \ set) \ list \Rightarrow _ \ \mathbf{where}$
 $mk_db \ t = Monitor.mk_db \ (\bigcup n \in \text{set} \ (\text{map} \ fst \ t). \ (\lambda v. \ (n, \ v)) \ ' \ the \ (\text{map_of} \ t \ n))$

definition $rbt_fold :: _ \Rightarrow event_data \ tuple \ set_rbt \Rightarrow _ \Rightarrow _ \ \mathbf{where}$
 $rbt_fold = RBT_Set2.fold$

definition $rbt_empty :: event_data \ list \ set_rbt \ \mathbf{where}$
 $rbt_empty = RBT_Set2.empty$

definition $rbt_insert :: _ \Rightarrow _ \Rightarrow event_data \ list \ set_rbt \ \mathbf{where}$
 $rbt_insert = RBT_Set2.insert$

lemma $saturate_commute:$
assumes $\bigwedge s. r \in g \ s \ \wedge \ s. g \ (\text{insert} \ r \ s) = g \ s \ \wedge \ s. r \in s \implies h \ s = g \ s$
and $terminates: \text{mono} \ g \ \wedge \ X. X \subseteq C \implies g \ X \subseteq C \ \text{finite} \ C$
shows $saturate \ g \ \{\} = saturate \ h \ \{r\}$
 $\langle proof \rangle$

definition $RPDs_aux = saturate \ (\lambda S. S \cup \bigcup (RPD \ ' \ S))$

lemma $RPDs_aux_code[code]:$
 $RPDs_aux \ S = (\text{let} \ S' = S \cup \text{Set.bind} \ S \ RPD \ \text{in} \ \text{if} \ S' \subseteq S \ \text{then} \ S \ \text{else} \ RPDs_aux \ S')$
 $\langle proof \rangle$

lemma $RPDs_code[code]: RPDs \ r = RPDs_aux \ \{r\}$
 $\langle proof \rangle$

definition $LPDs_aux = saturate \ (\lambda S. S \cup \bigcup (LPD \ ' \ S))$

lemma $LPDs_aux_code[code]:$
 $LPDs_aux \ S = (\text{let} \ S' = S \cup \text{Set.bind} \ S \ LPD \ \text{in} \ \text{if} \ S' \subseteq S \ \text{then} \ S \ \text{else} \ LPDs_aux \ S')$
 $\langle proof \rangle$

lemma $LPDs_code[code]: LPDs \ r = LPDs_aux \ \{r\}$

<proof>

lemma *is_empty_table_unfold* [code_unfold]:
 $X = \text{empty_table} \longleftrightarrow \text{Set.is_empty } X$
 $\text{empty_table} = X \longleftrightarrow \text{Set.is_empty } X$
 $\text{set_eq } X \text{ empty_table} \longleftrightarrow \text{Set.is_empty } X$
 $\text{set_eq empty_table } X \longleftrightarrow \text{Set.is_empty } X$
 $X = (\text{set_empty impl}) \longleftrightarrow \text{Set.is_empty } X$
 $(\text{set_empty impl}) = X \longleftrightarrow \text{Set.is_empty } X$
 $\text{set_eq } X (\text{set_empty impl}) \longleftrightarrow \text{Set.is_empty } X$
 $\text{set_eq } (\text{set_empty impl}) X \longleftrightarrow \text{Set.is_empty } X$
<proof>

lemma *tabulate_rbt_code*[code]: *Monitor.mrtabulate* (*xs* :: *mregex list*) *f* =
 (case ID CCOMPARE(*mregex*) of None \Rightarrow Code.abort (STR "tabulate RBT_Mapping: ccompare = None") ($\lambda_.$ Monitor.mrtabulate (*xs* :: *mregex list*) *f*)
 | $_ \Rightarrow$ RBT_Mapping (RBT_Mapping2.bulkload (List.map_filter ($\lambda k.$ let *fk* = *f k* in if *fk* = empty_table then None else Some (*k*, *fk*)) *xs*)))
<proof>

lemma *combine_Mapping*[code]:
 fixes *t* :: ('a :: ccompare, 'b) *mapping_rbt shows*
 Mapping.combine *f* (RBT_Mapping *t*) (RBT_Mapping *u*) =
 (case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "combine RBT_Mapping: ccompare = None")
 ($\lambda_.$ *Mapping.combine* *f* (RBT_Mapping *t*) (RBT_Mapping *u*))
 | Some $_ \Rightarrow$ RBT_Mapping (RBT_Mapping2.join ($\lambda_.$ *f*) *t u*))
<proof>

lemma *upd_set_empty*[simp]: *upd_set m f {}* = *m*
<proof>

lemma *upd_set_insert*[simp]: *upd_set m f (insert x A)* = *Mapping.update x (f x) (upd_set m f A)*
<proof>

lemma *upd_set_fold*:
 assumes *finite A*
 shows *upd_set m f A* = *Finite_Set.fold* ($\lambda a.$ *Mapping.update a (f a)*) *m A*
<proof>

lift_definition *upd_cfi* :: ('a \Rightarrow 'b) \Rightarrow ('a, ('a, 'b) *mapping*) *comp_fun_idem*
 is $\lambda f a m.$ *Mapping.update a (f a) m*
<proof>

lemma *upd_set_code*[code]:
 upd_set m f A = (if *finite A* then *set_fold_cfi* (*upd_cfi f*) *m A* else Code.abort (STR "upd_set: infinite")
 ($\lambda_.$ *upd_set m f A*))
<proof>

lemma *lexordp_eq_code*[code]: *lexordp_eq xs ys* \longleftrightarrow (case *xs* of [] \Rightarrow True
 | *x* # *xs* \Rightarrow (case *ys* of [] \Rightarrow False
 | *y* # *ys* \Rightarrow if *x* < *y* then True else if *x* > *y* then False else *lexordp_eq xs ys*))
<proof>

definition *filter_set m X t* = *Mapping.filter* (*filter_cond X m t*) *m*

declare *shift_end.simps*[folded *filter_set_def*, *code*]

lemma *upd_set'_empty*[simp]: *upd_set' m d f {}* = *m*

<proof>

lemma *upd_set'_insert*: $d = f d \implies (\bigwedge x. f (f x) = f x) \implies \text{upd_set}' m d f (\text{insert } x A) =$
(let $m' = (\text{upd_set}' m d f A)$ *in case* $\text{Mapping.lookup } m' x \text{ of None} \implies \text{Mapping.update } x d m'$
| Some $v \implies \text{Mapping.update } x (f v) m'$)
<proof>

lemma *upd_set'_aux1*: $\text{upd_set}' \text{Mapping.empty } d f \{b. b = k \vee (a, b) \in A\} =$
 $\text{Mapping.update } k d (\text{upd_set}' \text{Mapping.empty } d f \{b. (a, b) \in A\})$
<proof>

lemma *upd_set'_aux2*: $\text{Mapping.lookup } m k = \text{None} \implies \text{upd_set}' m d f \{b. b = k \vee (a, b) \in A\} =$
 $\text{Mapping.update } k d (\text{upd_set}' m d f \{b. (a, b) \in A\})$
<proof>

lemma *upd_set'_aux3*: $\text{Mapping.lookup } m k = \text{Some } v \implies \text{upd_set}' m d f \{b. b = k \vee (a, b) \in A\} =$
 $\text{Mapping.update } k (f v) (\text{upd_set}' m d f \{b. (a, b) \in A\})$
<proof>

lemma *upd_set'_aux4*: $k \notin \text{fst } A \implies \text{upd_set}' \text{Mapping.empty } d f \{b. (k, b) \in A\} = \text{Mapping.empty}$
<proof>

lemma *upd_nested_empty[simp]*: $\text{upd_nested } m d f \{\} = m$
<proof>

definition *upd_nested_step* :: $'c \Rightarrow ('c \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow ('a, ('b, 'c) \text{ mapping}) \text{ mapping} \Rightarrow$
 $('a, ('b, 'c) \text{ mapping}) \text{ mapping}$ **where**
 $\text{upd_nested_step } d f x m = (\text{case } x \text{ of } (k, k') \Rightarrow$
 $(\text{case } \text{Mapping.lookup } m k \text{ of Some } m' \Rightarrow$
 $(\text{case } \text{Mapping.lookup } m' k' \text{ of Some } v \Rightarrow \text{Mapping.update } k (\text{Mapping.update } k' (f v) m') m$
 $| None \Rightarrow \text{Mapping.update } k (\text{Mapping.update } k' d m') m)$
 $| None \Rightarrow \text{Mapping.update } k (\text{Mapping.update } k' d \text{Mapping.empty } m))$

lemma *upd_nested_insert*:
 $d = f d \implies (\bigwedge x. f (f x) = f x) \implies \text{upd_nested } m d f (\text{insert } x A) =$
 $\text{upd_nested_step } d f x (\text{upd_nested } m d f A)$
<proof>

definition *upd_nested_max_tstp* **where**
 $\text{upd_nested_max_tstp } m d X = \text{upd_nested } m d (\text{max_tstp } d) X$

lemma *upd_nested_max_tstp_fold*:
assumes *finite* X
shows $\text{upd_nested_max_tstp } m d X = \text{Finite_Set.fold } (\text{upd_nested_step } d (\text{max_tstp } d)) m X$
<proof>

lift_definition *upd_nested_max_tstp_cfi* ::
 $ts + tp \Rightarrow ('a \times 'b, ('a, ('b, ts + tp) \text{ mapping}) \text{ mapping}) \text{ comp_fun_idem}$
is $\lambda d. \text{upd_nested_step } d (\text{max_tstp } d)$
<proof>

lemma *upd_nested_max_tstp_code[code]*:
 $\text{upd_nested_max_tstp } m d X = (\text{if } \text{finite } X \text{ then } \text{set_fold_cfi } (\text{upd_nested_max_tstp_cfi } d) m X$
 $\text{else } \text{Code.abort } (\text{STR } \text{"upd_nested_max_tstp: infinite"}) (\lambda_. \text{upd_nested_max_tstp } m d X))$
<proof>

declare *add_new_mmuaux'_def*[*unfolded add_new_mmuaux.simps, folded upd_nested_max_tstp_def, code*]

lemma *filter_set_empty*[simp]: *filter_set m {} t = m*
 ⟨*proof*⟩

lemma *filter_set_insert*[simp]: *filter_set m (insert x A) t = (let m' = filter_set m A t in
 case Mapping.lookup m' x of Some u ⇒ if t = u then Mapping.delete x m' else m' | _ ⇒ m')*
 ⟨*proof*⟩

lemma *filter_set_fold*:
assumes *finite A*
shows *filter_set m A t = Finite_Set.fold (λa m.
 case Mapping.lookup m a of Some u ⇒ if t = u then Mapping.delete a m else m | _ ⇒ m) m A*
 ⟨*proof*⟩

lift_definition *filter_cfi* :: *'b ⇒ ('a, ('a, 'b) mapping) comp_fun_idem*
is *λt a m.*
case Mapping.lookup m a of Some u ⇒ if t = u then Mapping.delete a m else m | _ ⇒ m
 ⟨*proof*⟩

lemma *filter_set_code*[code]:
*filter_set m A t = (if finite A then set_fold_cfi (filter_cfi t) m A else Code.abort (STR "upd_set:
 infinite")) (λ_. filter_set m A t)*
 ⟨*proof*⟩

lemma *filter_Mapping*[code]:
fixes *t :: ('a :: ccompare, 'b) mapping_rbt* **shows**
Mapping.filter P (RBT_Mapping t) =
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "filter RBT_Mapping: ccompare = None")
(λ_. Mapping.filter P (RBT_Mapping t))
| Some _ ⇒ RBT_Mapping (RBT_Mapping2.filter (case_prod P) t))
 ⟨*proof*⟩

definition *filter_join pos X m = Mapping.filter (join_filter_cond pos X) m*

declare *join_mmsaux.simps*[folded *filter_join_def*, *code*]

lemma *filter_join_False_empty*: *filter_join False {} m = m*
 ⟨*proof*⟩

lemma *filter_join_False_insert*: *filter_join False (insert a A) m =*
filter_join False A (Mapping.delete a m)
 ⟨*proof*⟩

lemma *filter_join_False*:
assumes *finite A*
shows *filter_join False A m = Finite_Set.fold Mapping.delete m A*
 ⟨*proof*⟩

lift_definition *filter_not_in_cfi* :: *('a, ('a, 'b) mapping) comp_fun_idem* **is** *Mapping.delete*
 ⟨*proof*⟩

lemma *filter_join_code*[code]:
filter_join pos A m =
(if ¬pos ∧ finite A ∧ card A < Mapping.size m then set_fold_cfi filter_not_in_cfi m A
else Mapping.filter (join_filter_cond pos A) m)
 ⟨*proof*⟩

definition *set_minus* :: *'a set ⇒ 'a set ⇒ 'a set* **where**

$set_minus\ X\ Y = X - Y$

lift_definition *remove_cfi* :: ('a, 'a set) comp_fun_idem

is $\lambda b\ a.\ a - \{b\}$

<proof>

lemma *set_minus_finite*:

assumes *fin*: finite *Y*

shows $set_minus\ X\ Y = Finite_Set.fold\ (\lambda a\ X.\ X - \{a\})\ X\ Y$

<proof>

lemma *set_minus_code*[code]: $set_minus\ X\ Y =$

(if finite Y \wedge card Y < card X then set_fold_cfi remove_cfi X Y else X - Y)

<proof>

declare *bin_join.simps*[folded *set_minus_def*, code]

definition *remove_Union* **where**

$remove_Union\ A\ X\ B = A - (\bigcup x \in X.\ B\ x)$

lemma *remove_Union_finite*:

assumes finite *X*

shows $remove_Union\ A\ X\ B = Finite_Set.fold\ (\lambda x\ A.\ A - B\ x)\ A\ X$

<proof>

lift_definition *remove_Union_cfi* :: ('a \Rightarrow 'b set) \Rightarrow ('a, 'b set) comp_fun_idem **is** $\lambda B\ x\ A.\ A - B\ x$

<proof>

lemma *remove_Union_code*[code]: $remove_Union\ A\ X\ B =$

(if finite X then set_fold_cfi (remove_Union_cfi B) A X else A - ($\bigcup x \in X.\ B\ x$))

<proof>

lemma *tabulate_remdups*: $Mapping.tabulate\ xs\ f = Mapping.tabulate\ (remdups\ xs)\ f$

<proof>

lift_definition *clearjunk* :: (String.literal \times event_data list set) list \Rightarrow (String.literal, event_data list set list) alist **is**

$\lambda t.\ List.map_filter\ (\lambda(p, X).\ if\ X = \{\}\ then\ None\ else\ Some\ (p, [X]))\ (AList.clearjunk\ t)$

<proof>

lemma *map_filter_snd_map_filter*: $List.map_filter\ (\lambda(a, b).\ if\ P\ b\ then\ None\ else\ Some\ (f\ a\ b))\ xs =$

$map\ (\lambda(a, b).\ f\ a\ b)\ (filter\ (\lambda x.\ \neg\ P\ (snd\ x))\ xs)$

<proof>

lemma *mk_db_code_alist*:

$mk_db\ t = Assoc_List_Mapping\ (clearjunk\ t)$

<proof>

lemma *mk_db_code*[code]:

$mk_db\ t = Mapping.of_alist\ (List.map_filter\ (\lambda(p, X).\ if\ X = \{\}\ then\ None\ else\ Some\ (p, [X]))\ (AList.clearjunk\ t))$

<proof>

declare *New_max.genericJoin.simps*[folded *remove_Union_def*, code]

declare *New_max.wrapperGenericJoin_def*[folded *remove_Union_def*, code]

References

- [1] D. Basin, T. Dardinier, L. Heimes, S. Krstić, M. Raszyk, J. Schneider, and D. Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In N. Peltier and V. Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12166 of *LNCS*, pages 432–453. Springer, 2020.
- [2] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.
- [3] D. Basin, S. Krstić, and D. Traytel. Almost event-rate independent monitoring of metric dynamic logic. In S. K. Lahiri and G. Reger, editors, *RV 2017*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.
- [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.