

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

Thibault Dardinier Lukas Heimes Martin Raszyk Joshua Schneider
Dmitriy Traytel

February 6, 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	5
1.4	Linear ordering	7
1.4.1	Code setup	10
2	Event parameters	11
3	Regular expressions	12
4	Metric first-order dynamic logic	19
4.1	Formulas and satisfiability	19
4.1.1	Syntax	19
4.1.2	Future reach	23
4.1.3	Semantics	23
4.2	Past-only formulas	27
4.3	Safe formulas	28
4.4	Slicing traces	32
4.5	Translation to n-ary conjunction	35
5	Optimized relational join	41
5.1	Binary join	41
5.2	Multi-way join	43

6	Generic monitoring algorithm	52
6.1	Monitorable formulas	52
6.2	Handling regular expressions	56
6.2.1	LPD	59
6.2.2	RPD	61
6.3	The executable monitor	63
6.4	Verdict delay	72
6.5	Specification	86
6.6	Correctness	87
6.6.1	Invariants	87
6.6.2	Initialisation	92
6.6.3	Evaluation	95
6.6.4	Monitor step	149
6.6.5	Monitor function	149
6.7	Collected correctness results	152
7	Efficient implementation of temporal operators	152
7.1	Optimized queue data structure	152
7.2	Optimized data structure for Since	156
7.3	Optimized data structure for Until	178
8	Instantiation of the generic algorithm and code setup	205

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**
 $\lambda(_, e::'e \text{ word}, f::'f \text{ word}) (s::1 \text{ word}, _, _). (s, e, f)$.

lemma *is_nan_copysign[simp]*: *is_nan* (*copysign* *x y*) \longleftrightarrow *is_nan* *x*
unfolding *is_nan_def* **by** *transfer auto*

1.2 Additional lemmas about generic floats

lemma *is_nan_some_nan[simp]*: *is_nan* (*some_nan* :: ('e, 'f) float)
unfolding *some_nan_def*
by (*rule* *someI*[**where** *x*=*Abs_float* (0, - 1 :: 'e word, 1)])
(*auto simp add: is_nan_def exponent_def fraction_def emax_def Abs_float_inverse[simplified]*)

lemma *not_is_nan_0[simp]*: \neg *is_nan* 0
unfolding *is_nan_def* **by** (*simp add: zero_simps*)

lemma *not_is_nan_1[simp]*: \neg *is_nan* 1
unfolding *is_nan_def* **by** *transfer simp*

lemma *is_nan_plus*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* + *y*)
unfolding *plus_float_def fadd_def* **by** *auto*

lemma *is_nan_minus*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* - *y*)
unfolding *minus_float_def fsub_def* **by** *auto*

lemma *is_nan_times*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* * *y*)
unfolding *times_float_def fmul_def* **by** *auto*

lemma *is_nan_divide*: *is_nan* *x* \vee *is_nan* *y* \Longrightarrow *is_nan* (*x* / *y*)

unfolding *divide_float_def fdiv_def* **by** *auto*

lemma *is_nan_float_sqrt*: $is_nan\ x \implies is_nan\ (float_sqrt\ x)$
unfolding *float_sqrt_def fsqrt_def* **by** *simp*

lemma *nan_fcompare*: $is_nan\ x \vee is_nan\ y \implies fcompare\ x\ y = Und$
unfolding *fcompare_def* **by** *simp*

lemma *nan_not_le*: $is_nan\ x \vee is_nan\ y \implies \neg x \leq y$
unfolding *less_eq_float_def fle_def fcompare_def* **by** *simp*

lemma *nan_not_less*: $is_nan\ x \vee is_nan\ y \implies \neg x < y$
unfolding *less_float_def flt_def fcompare_def* **by** *simp*

lemma *nan_not_zero*: $is_nan\ x \implies \neg is_zero\ x$
unfolding *is_nan_def is_zero_def* **by** *simp*

lemma *nan_not_infinity*: $is_nan\ x \implies \neg is_infinity\ x$
unfolding *is_nan_def is_infinity_def* **by** *simp*

lemma *zero_not_infinity*: $is_zero\ x \implies \neg is_infinity\ x$
unfolding *is_zero_def is_infinity_def* **by** *simp*

lemma *zero_not_nan*: $is_zero\ x \implies \neg is_nan\ x$
unfolding *is_zero_def is_nan_def* **by** *simp*

lemma *minus_one_power_one_word*: $(-1 :: real) ^{unat\ (x :: 1\ word)} = (if\ unat\ x = 0\ then\ 1\ else\ -1)$
proof –
have $unat\ x = 0 \vee unat\ x = 1$
using *le_Suc[OF unat_one_word_le]* **by** *auto*
then show *?thesis* **by** *auto*
qed

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

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

lemma *valof_alt*: $valof\ x = (if\ exponent\ x = 0\ then\ if\ sign\ x = 0\ then\ valofd\ x\ else\ -\ valofd\ x\ else\ if\ sign\ x = 0\ then\ valofn\ x\ else\ -\ valofn\ x)$
unfolding *valofn_def valofd_def*
by *transfer (auto simp: minus_one_power_one_word unat_eq_0 field_simps)*

lemma *fraction_less_2p*: $fraction\ (x :: ('e, 'f)\ float) < 2^{LENGTH ('f)}$
by *transfer auto*

lemma *valofn_ge_0*: $0 \leq valofn\ x$
unfolding *valofn_def* **by** *simp*

lemma *valofn_ge_2p*: $2^{exponent\ (x :: ('e, 'f)\ float)} / 2^{bias\ TYPE (('e, 'f)\ float)} \leq valofn\ x$
unfolding *valofn_def* **by** *(simp add: field_simps)*

lemma *valofn_less_2p*:
fixes $x :: ('e, 'f)\ float$

```

assumes exponent  $x < e$ 
shows  $\text{valofn } x < 2^e / 2^{\text{bias TYPE}((e, f) \text{ float})}$ 
proof -
  have  $1 + \text{real } (\text{fraction } x) / 2^{\text{LENGTH}(f)} < 2$ 
    by (simp add: fraction_less_2p)
  then have  $\text{valofn } x < (2^{\text{exponent } x} / 2^{\text{bias TYPE}((e, f) \text{ float})}) * 2$ 
    unfolding valofn_def by (simp add: field_simps)
  also have  $\dots \leq 2^e / 2^{\text{bias TYPE}((e, f) \text{ float})}$ 
    using assms by (auto simp: less_eq_Suc_le field_simps elim: order_trans[rotated, OF exp_less])
  finally show ?thesis .
qed

```

```

lemma valofd_ge_0:  $0 \leq \text{valofd } x$ 
  unfolding valofd_def by simp

```

```

lemma valofd_less_2p:  $\text{valofd } (x :: (e, f) \text{ float}) < 2 / 2^{\text{bias TYPE}((e, f) \text{ float})}$ 
  unfolding valofd_def
  by (simp add: fraction_less_2p field_simps)

```

```

lemma valofn_le_imp_exponent_le:
  fixes  $x \ y :: (e, f) \text{ float}$ 
  assumes  $\text{valofn } x \leq \text{valofn } y$ 
  shows  $\text{exponent } x \leq \text{exponent } y$ 
proof (rule ccontr)
  assume  $\neg \text{exponent } x \leq \text{exponent } y$ 
  then have  $\text{valofn } y < 2^{\text{exponent } x} / 2^{\text{bias TYPE}((e, f) \text{ float})}$ 
    using valofn_less_2p[of y] by auto
  also have  $\dots \leq \text{valofn } x$  by (rule valofn_ge_2p)
  finally show False using assms by simp
qed

```

```

lemma valofn_eq:
  fixes  $x \ y :: (e, f) \text{ float}$ 
  assumes  $\text{valofn } x = \text{valofn } y$ 
  shows  $\text{exponent } x = \text{exponent } y \ \text{fraction } x = \text{fraction } y$ 
proof -
  from assms show  $\text{exponent } x = \text{exponent } y$ 
    by (auto intro!: antisym valofn_le_imp_exponent_le)
  with assms show  $\text{fraction } x = \text{fraction } y$ 
    unfolding valofn_def by (simp add: field_simps)
qed

```

```

lemma valofd_eq:
  fixes  $x \ y :: (e, f) \text{ float}$ 
  assumes  $\text{valofd } x = \text{valofd } y$ 
  shows  $\text{fraction } x = \text{fraction } y$ 
  using assms unfolding valofd_def by (simp add: field_simps)

```

```

lemma is_zero_valof_conv:  $\text{is\_zero } x \longleftrightarrow \text{valof } x = 0$ 
  unfolding is_zero_def valof_alt
  using valofn_ge_2p[of x] by (auto simp: valofd_def field_simps dest: order.antisym)

```

```

lemma valofd_neq_valofn:
  fixes  $x \ y :: (e, f) \text{ float}$ 
  assumes  $\text{exponent } y \neq 0$ 
  shows  $\text{valofd } x \neq \text{valofn } y \ \text{valofn } y \neq \text{valofd } x$ 
proof -
  have  $\text{valofd } x < 2 / 2^{\text{bias TYPE}((e, f) \text{ float})}$  by (rule valofd_less_2p)

```

```

also have ...  $\leq 2^{\wedge} IEEE.exponent\ y / 2^{\wedge} bias\ TYPE('e, 'f)\ IEEE.float$ 
  using assms by (simp add: self_le_power_field_simps)
also have ...  $\leq valofn\ y$  by (rule valofn_ge_2p)
finally show valofd x  $\neq valofn\ y$  valofn y  $\neq valofd\ x$  by simp_all
qed

```

```

lemma sign_gt_0_conv:  $0 < sign\ x \longleftrightarrow sign\ x = 1$ 
  by (cases x rule: sign_cases) simp_all

```

```

lemma valof_eq:
  assumes  $\neg is\_zero\ x \vee \neg is\_zero\ y$ 
  shows valof x = valof y  $\longleftrightarrow x = y$ 
proof
  assume *: valof x = valof y
  with assms have valof y  $\neq 0$  by (simp add: is_zero_valof_conv)
  with * show x = y
    unfolding valof_alt
    using valofd_ge_0[of x] valofd_ge_0[of y] valofn_ge_0[of x] valofn_ge_0[of y]
    by (auto simp: valofd_neq_valofn sign_gt_0_conv split: if_splits
      intro!: float_eqI elim: valofn_eq valofd_eq)
qed simp

```

```

lemma zero_fcompare:  $is\_zero\ x \implies is\_zero\ y \implies fcompare\ x\ y = ccode.Eq$ 
  unfolding fcompare_def by (simp add: zero_not_infinity zero_not_nan is_zero_valof_conv)

```

1.3 Doubles with a unified NaN value

```

quotient_type double = (11, 52) float /  $\lambda x\ y. is\_nan\ x \wedge is\_nan\ y \vee x = y$ 
  by (auto intro!: equivpI reflpI sympI transpI)

```

```

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

```

```

lift_definition zero_double :: double is 0 .

```

```

lift_definition one_double :: double is 1 .

```

```

lift_definition plus_double :: double  $\Rightarrow$  double  $\Rightarrow$  double is plus
  by (auto simp: is_nan_plus)

```

```

lift_definition minus_double :: double  $\Rightarrow$  double  $\Rightarrow$  double is minus
  by (auto simp: is_nan_minus)

```

```

lift_definition uminus_double :: double  $\Rightarrow$  double is uminus
  by auto

```

```

lift_definition times_double :: double  $\Rightarrow$  double  $\Rightarrow$  double is times
  by (auto simp: is_nan_times)

```

```

lift_definition less_eq_double :: double  $\Rightarrow$  double  $\Rightarrow$  bool is ( $\leq$ )
  by (auto simp: nan_not_le)

```

```

lift_definition less_double :: double  $\Rightarrow$  double  $\Rightarrow$  bool is ( $<$ )
  by (auto simp: nan_not_less)

```

```

instance ..

```

```

end

```

```

instantiation double :: inverse
begin

lift_definition divide_double :: double ⇒ double ⇒ double is divide
  by (auto simp: is_nan_divide)

definition inverse_double :: double ⇒ double where
  inverse_double x = 1 div x

instance ..

end

lift_definition sqrt_double :: double ⇒ double is float_sqrt
  by (auto simp: is_nan_float_sqrt)

no_notation plus_infinity (⟨∞⟩)

lift_definition infinity :: double is plus_infinity .

lift_definition nan :: double is some_nan .

lift_definition is_zero :: double ⇒ bool is IEEE.is_zero
  by (auto simp: nan_not_zero)

lift_definition is_infinite :: double ⇒ bool is IEEE.is_infinity
  by (auto simp: nan_not_infinity)

lift_definition is_nan :: double ⇒ bool is IEEE.is_nan
  by auto

lemma is_nan_conv: is_nan x ⟷ x = nan
  by transfer auto

lift_definition copysign_double :: double ⇒ double ⇒ double is
  λx y. if IEEE.is_nan y then some_nan else copysign x y
  by auto

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

lift_definition fcompare_double :: double ⇒ double ⇒ ccode is fcompare
  by (auto simp: nan_fcompare)

lemma nan_fcompare_double: is_nan x ∨ is_nan y ⟹ fcompare_double x y = Und
  by transfer (rule nan_fcompare)

consts compare_double :: double ⇒ double ⇒ integer

specification (compare_double)
  compare_double_less: compare_double x y < 0 ⟷ is_nan x ∧ ¬ is_nan y ∨ fcompare_double x y =
  ccode.Lt
  compare_double_eq: compare_double x y = 0 ⟷ is_nan x ∧ is_nan y ∨ fcompare_double x y =
  ccode.Eq
  compare_double_greater: compare_double x y > 0 ⟷ ¬ is_nan x ∧ is_nan y ∨ fcompare_double x
  y = ccode.Gt
  by (rule exI[where x=λx y. if is_nan x then if is_nan y then 0 else -1
  else if is_nan y then 1 else (case fcompare_double x y of ccode.Eq ⇒ 0 | ccode.Lt ⇒ -1 | ccode.Gt
  ⇒ 1)],

```

transfer, auto simp: fcompare_def)

lemmas *compare_double_simps* = *compare_double_less compare_double_eq compare_double_greater*

lemma *compare_double_le_0*: *compare_double x y ≤ 0* \longleftrightarrow
is_nan x \vee *fcompare_double x y* \in {*ccode.Eq*, *ccode.Lt*}
by (*rule linorder_cases*[*of compare_double x y 0*]; *simp*)
(*auto simp: compare_double_simps nan_fcompare_double*)

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

definition *double_of_int* **where** [*code del*]: *double_of_int x* = *double_of_integer* (*integer_of_int x*)

lemma [*code*]: *double_of_int* (*int_of_integer x*) = *double_of_integer x*
unfolding *double_of_int_def* **by** *simp*

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*)]
by *auto*

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* \longleftrightarrow *fcompare_double y x* = *ccode.Lt*
by *transfer* (*auto simp: fcompare_def*)

lemma *fcompare_double_refl*: \neg *is_nan x* \implies *fcompare_double x x* = *ccode.Eq*
by *transfer* (*auto simp: fcompare_def*)

lemma *fcompare_double_Eq1*: *fcompare_double x y* = *ccode.Eq* \implies *fcompare_double y z* = *c* \implies *fcompare_double x z* = *c*
by *transfer* (*auto simp: fcompare_def split: if_splits*)

lemma *fcompare_double_Eq2*: *fcompare_double y z* = *ccode.Eq* \implies *fcompare_double x y* = *c* \implies *fcompare_double x z* = *c*
by *transfer* (*auto simp: fcompare_def split: if_splits*)

lemma *fcompare_double_Lt_trans*: *fcompare_double x y* = *ccode.Lt* \implies *fcompare_double y z* = *ccode.Lt*
 \implies *fcompare_double x z* = *ccode.Lt*
by *transfer* (*auto simp: fcompare_def split: if_splits*)

lemma *fcompare_double_eq*: \neg *is_zero x* \vee \neg *is_zero y* \implies *fcompare_double x y* = *ccode.Eq* \implies *x* = *y*
by *transfer* (*auto simp: fcompare_def valof_eq IEEE.is_infinity_def split: if_splits intro!: float_eqI*)

lemma *fcompare_double_Lt_asym*: *fcompare_double x y* = *ccode.Lt* \implies *fcompare_double y x* = *ccode.Lt*
 \implies *False*
by *transfer* (*auto simp: fcompare_def split: if_splits*)

lemma *compare_double_swap*: $0 <$ *compare_double x y* \longleftrightarrow *compare_double y x* < 0
by (*auto simp: compare_double_simps fcompare_double_swap*)

lemma *compare_double_refl*: $\text{compare_double } x \ x = 0$
by (*auto simp: compare_double_eq intro!: fcompare_double_refl*)

lemma *compare_double_trans*: $\text{compare_double } x \ y \leq 0 \implies \text{compare_double } y \ z \leq 0 \implies \text{compare_double } x \ z \leq 0$
by (*fastforce simp: compare_double_le_0 nan_fcompare_double*
dest: fcompare_double_Eq1 fcompare_double_Eq2 fcompare_double_Lt_trans)

lemma *compare_double_antisym*: $\text{compare_double } x \ y \leq 0 \implies \text{compare_double } y \ x \leq 0 \implies \neg \text{is_zero } x \vee \neg \text{is_zero } y \implies x = y$
unfolding *compare_double_le_0*
by (*auto simp: nan_fcompare_double is_nan_conv*
intro: fcompare_double_eq fcompare_double_eq[symmetric]
dest: fcompare_double_Lt_asym)

lemma *zero_compare_double_copysign*: $\text{compare_double } (\text{copysign_double } 1 \ x) \ (\text{copysign_double } 1 \ y) \leq 0 \implies \text{is_zero } x \implies \text{is_zero } y \implies \text{compare_double } x \ y \leq 0$
unfolding *compare_double_le_0*
by *transfer (auto simp: nan_not_zero zero_fcompare split: if_splits)*

lemma *is_zero_double_cases*: $\text{is_zero } x \implies (x = 0 \implies P) \implies (x = -0 \implies P) \implies P$
by *transfer (auto elim!: is_zero_cases)*

lemma *copysign_1_0[simp]*: $\text{copysign_double } 1 \ 0 = 1 \ \text{copysign_double } 1 \ (-0) = -1$
by (*transfer, simp, transfer, auto*)⁺

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

lemma *not_is_zero_one_double[simp]*: $\neg \text{is_zero } 1$
by (*transfer, unfold IEEE.is_zero_def, transfer, simp*)

lemma *uminus_one_neq_one_double[simp]*: $-1 \neq (1 :: \text{double})$
by (*transfer, transfer, simp*)

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

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

lemma *lcompare_double_ge_0*: $\text{lcompare_double } x \ y \geq 0 \longleftrightarrow \text{lle_double } y \ x$
unfolding *lle_double_def lcompare_double_def*
using *compare_double_swap not_less* **by** *auto*

lemma *lcompare_double_gt_0*: $\text{lcompare_double } x \ y > 0 \longleftrightarrow \text{lless_double } y \ x$
unfolding *lless_double_def lcompare_double_def*
using *compare_double_swap* **by** *auto*

lemma *lcompare_double_eq_0*: $\text{lcompare_double } x \ y = 0 \longleftrightarrow x = y$
proof
assume *: $\text{lcompare_double } x \ y = 0$
show $x = y$ **proof** (*cases is_zero x ∧ is_zero y*)
case *True*
with * **show** *?thesis*
by (*fastforce simp: lcompare_double_def compare_double_simps is_nan_conv*)

```

      elim: is_zero_double_cases dest!: fcompare_double_eq[rotated])
next
  case False
  with * show ?thesis
    by (auto simp: lcompare_double_def linorder_not_less[symmetric] compare_double_swap
        intro!: compare_double_antisym)
qed
next
  assume x = y
  then show lcompare_double x y = 0
    by (simp add: lcompare_double_def compare_double_refl)
qed

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
  fix x y z :: double
  show lless_double x y  $\longleftrightarrow$  lle_double x y  $\wedge$   $\neg$  lle_double y x
    unfolding lless_double_def lle_double_def lcompare_double_def
    by (auto simp: compare_double_swap not_le)
  show lle_double x x
    unfolding lle_double_def lcompare_double_def
    by (auto simp: compare_double_refl)
  show lle_double x z if lle_double x y and lle_double y z
    using that
    by (auto 0 3 simp: lle_double_def lcompare_double_def not_le compare_double_swap
        split: if_splits dest: compare_double_trans zero_compare_double_copysign
        zero_compare_double_copysign[OF less_imp_le] compare_double_antisym)
  show x = y if lle_double x y and lle_double y x
  proof (cases is_zero x  $\wedge$  is_zero y)
    case True
    with that show ?thesis
      by (auto 0 3 simp: lle_double_def lcompare_double_def elim: is_zero_double_cases
          dest!: compare_double_antisym)
    next
    case False
    with that show ?thesis
      by (auto simp: lle_double_def lcompare_double_def elim!: compare_double_antisym)
  qed
  show lle_double x y  $\vee$  lle_double y x
    unfolding lle_double_def lcompare_double_def
    by (auto simp: compare_double_swap not_le)
  qed

instantiation double :: equal
begin

definition equal_double :: double  $\Rightarrow$  double  $\Rightarrow$  bool where
  equal_double x y  $\longleftrightarrow$  lcompare_double x y = 0

instance by intro_classes (simp add: equal_double_def lcompare_double_eq_0)

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
  unfolding comparator_double_def
  by (auto simp: lcompare_double_0_folds split: if_splits intro!: comparator.intro)

local_setup <
  Comparator_Generator.register_foreign_comparator @{typ double}
  @{term comparator_double}
  @{thm comparator_double}
>

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

unfolding div_to_zero_def mod_to_zero_def Let_def

by (auto simp: minus_mod_eq_mult_div[symmetric] div_minus_right mod_minus_right ac_simps)

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 ⇒ event_data ⇒ bool **where**

```

less_event_data x y ↔ x ≤ y ∧ ¬ y ≤ x

```

fun plus_event_data **where**

```

  EInt x + EInt y = EInt (x + y)
| 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 ..

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)
by (induct r) auto

definition Wild = Skip 1

lemma *size_regex_estimation*[*termination_simp*]: $x \in \text{atms } r \implies y < f x \implies y < \text{size_regex } f r$
by (*induct r*) *auto*

lemma *size_regex_estimation'*[*termination_simp*]: $x \in \text{atms } r \implies y \leq f x \implies y \leq \text{size_regex } f r$
by (*induct r*) *auto*

qualified definition *TimesL* $r S = \text{Times } r \text{ ' } S$

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

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*]:

$r = r' \implies (\bigwedge z. z \in \text{atms } r \implies \text{fv } z = \text{fv}' z) \implies \text{fv_regex } f v r = \text{fv_regex } f v' r'$
by (*induct r arbitrary: r'*) *auto*

lemma *finite_fv_regex*[*simp*]: $(\bigwedge z. z \in \text{atms } r \implies \text{finite } (f v z)) \implies \text{finite } (\text{fv_regex } f v r)$
by (*induct r*) *auto*

lemma *fv_regex_commute*:

$(\bigwedge z. z \in \text{atms } r \implies x \in \text{fv } z \iff g x \in \text{fv}' z) \implies x \in \text{fv_regex } f v r \iff g x \in \text{fv_regex } f v' r$
by (*induct r*) *auto*

lemma *fv_regex_alt*: $\text{fv_regex } f v r = (\bigcup z \in \text{atms } r. \text{fv } z)$
by (*induct r*) *auto*

qualified definition *nfv_regex* **where**

nfv_regex *fv* *r* = *Max* (*insert* 0 (*Suc* ' *fv_regex* *fv* *r*))

lemma *insert_Un*: $\text{insert } x (A \cup B) = \text{insert } x A \cup \text{insert } x B$
by *auto*

lemma *nfv_regex_simps*[*simp*]:

assumes [*simp*]: $(\bigwedge z. z \in \text{atms } r \implies \text{finite } (f v z)) (\bigwedge z. z \in \text{atms } s \implies \text{finite } (f v z))$

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*

unfolding *nfv_regex_def*

by (*auto simp add: image_Un Max_Un insert_Un simp del: Un_insert_right Un_insert_left*)

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

qualified primrec *match* :: $(\text{nat} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ regex} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

match test (*Skip* *n*) = $(\lambda i j. j = i + n)$
| *match test* (*Test* φ) = $(\lambda i j. i = j \wedge \text{test } i \varphi)$
| *match test* (*Plus* *r s*) = *match test* *r* \sqcup *match test* *s*
| *match test* (*Times* *r s*) = *match test* *r* *OO* *match test* *s*
| *match test* (*Star* *r*) = (*match test* *r*)**

lemma *match_cong*[*fundef_cong*]:

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

by (induct r arbitrary: r') auto

qualified primrec eps where

eps test i (Skip n) = (n = 0)
| eps test i (Test φ) = test i φ
| eps test i (Plus r s) = (eps test i r \vee eps test i s)
| eps test i (Times r s) = (eps test i r \wedge eps test i s)
| eps test i (Star r) = True

qualified primrec lpd where

lpd test i (Skip n) = (case n of 0 \Rightarrow {} | Suc m \Rightarrow {Skip m})
| lpd test i (Test φ) = {}
| lpd test i (Plus r s) = (lpd test i r \cup lpd test i s)
| lpd test i (Times r s) = TimesR (lpd test i r) s \cup (if eps test i r then lpd test i s else {})
| lpd test i (Star r) = TimesR (lpd test i r) (Star r)

qualified primrec lpd κ where

lpd κ κ test i (Skip n) = (case n of 0 \Rightarrow {} | Suc m \Rightarrow { κ (Skip m)})
| lpd κ κ test i (Test φ) = {}
| lpd κ κ test i (Plus r s) = lpd κ κ test i r \cup lpd κ κ test i s
| lpd κ κ test i (Times r s) = lpd κ ($\lambda t. \kappa$ (Times t s)) test i r \cup (if eps test i r then lpd κ κ test i s else {})
| lpd κ κ test i (Star r) = lpd κ ($\lambda t. \kappa$ (Times t (Star r))) test i r

qualified primrec rpd where

rpd test i (Skip n) = (case n of 0 \Rightarrow {} | Suc m \Rightarrow {Skip m})
| rpd test i (Test φ) = {}
| rpd test i (Plus r s) = (rpd test i r \cup rpd test i s)
| rpd test i (Times r s) = TimesL r (rpd test i s) \cup (if eps test i s then rpd test i r else {})
| rpd test i (Star r) = TimesL (Star r) (rpd test i r)

qualified primrec rpd κ where

rpd κ κ test i (Skip n) = (case n of 0 \Rightarrow {} | Suc m \Rightarrow { κ (Skip m)})
| rpd κ κ test i (Test φ) = {}
| rpd κ κ test i (Plus r s) = rpd κ κ test i r \cup rpd κ κ test i s
| rpd κ κ test i (Times r s) = rpd κ ($\lambda t. \kappa$ (Times r t)) test i s \cup (if eps test i s then rpd κ κ test i r else {})
| rpd κ κ test i (Star r) = rpd κ ($\lambda t. \kappa$ (Times (Star r) t)) test i r

lemma lpd κ _lpd: lpd κ κ test i r = κ ' lpd test i r

by (induct r arbitrary: κ) (auto simp: TimesR_def split: nat.splits)

lemma rpd κ _rpd: rpd κ κ test i r = κ ' rpd test i r

by (induct r arbitrary: κ) (auto simp: TimesL_def split: nat.splits)

lemma match_le: match test r i j \Longrightarrow i \leq j

proof (induction r arbitrary: i j)

case (Times r s)

then show ?case using order.trans by fastforce

next

case (Star r)

from Star.premis show ?case

unfolding match.simps by (induct i j rule: rtranclp.induct) (force dest: Star.IH)+

qed auto

lemma match_rtranclp_le: (match test r)** i j \Longrightarrow i \leq j

by (metis match.simps(5) match_le)

lemma eps_match: eps test i r \longleftrightarrow match test r i i

by (induction r) (auto dest: antisym[OF match_le match_le])

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 (induction r arbitrary: i j)
 case (Times r s)
 have match test (Times r s) i j $\longleftrightarrow (\exists k. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j)$
 by auto
 also have ... $\longleftrightarrow \text{match test } r \ i \ i \wedge \text{match test } s \ i \ j \vee$
 ($\exists k > i. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j$)
 using match_le[of test r i] nat_less_le by auto
 also have ... $\longleftrightarrow \text{match test } r \ i \ i \wedge (\bigsqcup t \in \text{lpd test } i \ s. \text{match test } t) \ (i + 1) \ j \vee$
 ($\exists k > i. (\bigsqcup t \in \text{lpd test } i \ r. \text{match test } t) \ (i + 1) \ k \wedge \text{match test } s \ k \ j$)
 using Times.IH(1) Times.IH(2)[OF Times.prem] by metis
 also have ... $\longleftrightarrow \text{match test } r \ i \ i \wedge (\bigsqcup t \in \text{lpd test } i \ s. \text{match test } t) \ (i + 1) \ j \vee$
 ($\exists k. (\bigsqcup t \in \text{lpd test } i \ r. \text{match test } t) \ (i + 1) \ k \wedge \text{match test } s \ k \ j$)
 using Times.prem by (intro disj_cong[OF refl] iff_exI) (auto dest: match_le)
 also have ... $\longleftrightarrow (\bigsqcup (\text{match test } \text{'lpd test } i \ (\text{Times } r \ s))) \ (i + 1) \ j$
 by (force simp: TimesL_def TimesR_def eps_match)
 finally show ?case .

next
 case (Star r)
 have $\exists s \in \text{lpd test } i \ r. (\text{match test } s \ \text{OO} \ (\text{match test } r)^{**}) \ (i + 1) \ j$ if (match test r)** i j
 using that Star.prem match_le[of test _ i + 1]
 proof (induct rule: converse_rtranclp_induct)
 case (step i k)
 then show ?case
 by (cases i < k) (auto simp: not_less Star.IH dest: match_le)
 qed simp
 with Star.prem show ?case using match_le[of test _ i + 1]
 by (auto simp: TimesL_def TimesR_def Suc_le_eq Star.IH[of i]
 elim!: converse_rtranclp_into_rtranclp[rotated])
 qed (auto split: nat.splits)

lemma *rpj_match*: $i < j \implies \text{match test } r \ i \ j \longleftrightarrow (\bigsqcup s \in \text{rpj test } j \ r. \text{match test } s) \ i \ (j - 1)$

proof (induction r arbitrary: i j)
 case (Times r s)
 have match test (Times r s) i j $\longleftrightarrow (\exists k. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j)$
 by auto
 also have ... $\longleftrightarrow \text{match test } r \ i \ j \wedge \text{match test } s \ j \ j \vee$
 ($\exists k < j. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j$)
 using match_le[of test s _ j] nat_less_le by auto
 also have ... $\longleftrightarrow (\bigsqcup t \in \text{rpj test } j \ r. \text{match test } t) \ i \ (j - 1) \wedge \text{match test } s \ j \ j \vee$
 ($\exists k < j. \text{match test } r \ i \ k \wedge (\bigsqcup t \in \text{rpj test } j \ s. \text{match test } t) \ k \ (j - 1)$)
 using Times.IH(1)[OF Times.prem] Times.IH(2) by metis
 also have ... $\longleftrightarrow (\bigsqcup t \in \text{rpj test } j \ r. \text{match test } t) \ i \ (j - 1) \wedge \text{match test } s \ j \ j \vee$
 ($\exists k. \text{match test } r \ i \ k \wedge (\bigsqcup t \in \text{rpj test } j \ s. \text{match test } t) \ k \ (j - 1)$)
 using Times.prem by (intro disj_cong[OF refl] iff_exI) (auto dest: match_le)
 also have ... $\longleftrightarrow (\bigsqcup (\text{match test } \text{'rpj test } j \ (\text{Times } r \ s))) \ i \ (j - 1)$
 by (force simp: TimesL_def TimesR_def eps_match)
 finally show ?case .

next
 case (Star r)
 have $\exists s \in \text{rpj test } j \ r. ((\text{match test } r)^{**} \ \text{OO} \ \text{match test } s) \ i \ (j - 1)$ if (match test r)** i j
 using that Star.prem match_le[of test _ _ j - 1]
 proof (induct rule: rtranclp_induct)
 case (step k j)
 then show ?case
 by (cases k < j) (auto simp: not_less Star.IH dest: match_le)

qed *simp*
with *Star.premis show ?case*
by (*auto 0 3 simp: TimesL_def TimesR_def intro: Star.IH[of _ j, THEN iffD2]*
elim!: rtranclp.rtrancl_into_rtrancl dest: match_le[of test __ j - 1, unfolded One_nat_def])
qed (*auto split: nat.splits*)

lemma *lpd_fv_regex*: $s \in \text{lpd test } i \ r \implies \text{fv_regex } \text{fv } s \subseteq \text{fv_regex } \text{fv } r$
by (*induct r arbitrary: s*) (*auto simp: TimesR_def TimesL_def split: if_splits nat.splits*)⁺

lemma *rpd_fv_regex*: $s \in \text{rpd test } i \ r \implies \text{fv_regex } \text{fv } s \subseteq \text{fv_regex } \text{fv } r$
by (*induct r arbitrary: s*) (*auto simp: TimesR_def TimesL_def split: if_splits nat.splits*)⁺

lemma *match_fv_cong*:
 $(\bigwedge i \ x. x \in \text{atms } r \implies \text{test } i \ x = \text{test}' i \ x) \implies \text{match test } r = \text{match test}' r$
by (*induct r*) *auto*

lemma *eps_fv_cong*:
 $(\bigwedge i \ x. x \in \text{atms } r \implies \text{test } i \ x = \text{test}' i \ x) \implies \text{eps test } i \ r = \text{eps test}' i \ r$
by (*induct r*) *auto*

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) =*
 $((g = Lax \vee \text{fv_regex } \text{fv } r \subseteq \text{fv_regex } \text{fv } s) \wedge \text{safe_regex } \text{Futu } g \ s \wedge \text{safe_regex } \text{Futu } \text{Lax } r)$
| *safe_regex Past g (Times r s) =*
 $((g = Lax \vee \text{fv_regex } \text{fv } s \subseteq \text{fv_regex } \text{fv } r) \wedge \text{safe_regex } \text{Past } g \ r \wedge \text{safe_regex } \text{Past } \text{Lax } s)$
| *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*:
 $(\bigwedge x. x \in \text{atms } r \implies \text{safe } \text{Strict } x \implies \text{safe } \text{Lax } x) \implies \text{safe_regex } m \ \text{Strict } r \implies \text{safe_regex } m \ \text{Lax } r$
by (*induct r; cases m*) *auto*

lemma *safe_lpd_fv_regex_le*: $\text{safe_regex } \text{Futu } \text{Strict } r \implies s \in \text{lpd test } i \ r \implies \text{fv_regex } \text{fv } r \subseteq \text{fv_regex } \text{fv } s$
by (*induct r*) (*auto simp: TimesR_def split: if_splits*)

lemma *safe_lpd_fv_regex*: $\text{safe_regex } \text{Futu } \text{Strict } r \implies s \in \text{lpd test } i \ r \implies \text{fv_regex } \text{fv } s = \text{fv_regex } \text{fv } r$
by (*simp add: eq_iff_lpd_fv_regex safe_lpd_fv_regex_le*)

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

proof (*induct r arbitrary: s*)
case (*Plus r1 r2*)
from *Plus(3,4) show ?case*
by (*auto elim: Plus(1,2)*)

```

next
  case (Times r1 r2)
  from Times(3,4) show ?case
  by (auto simp: TimesR_def elim: Times(1,2) split: if_splits)
qed (auto simp: TimesR_def split: nat.splits)

lemma safe_lpd: ( $\forall x \in \text{atms } r. \text{safe Strict } x \longrightarrow \text{safe Lax } x$ )  $\implies$ 
  safe_regex Futu Strict  $r \implies s \in \text{lpd test } i \ r \implies \text{safe\_regex Futu Strict } s$ 
proof (induct r arbitrary: s)
  case (Plus r1 r2)
  from Plus(3,4,5) show ?case
  by (auto elim: Plus(1,2) simp: ball_Un)
next
  case (Times r1 r2)
  from Times(3,4,5) show ?case
  by (force simp: TimesR_def ball_Un elim: Times(1,2) cosafe_lpd dest: lpd_fv_regex split: if_splits)
next
  case (Star r)
  from Star(2,3,4) show ?case
  by (force simp: TimesR_def elim: Star(1) cosafe_lpd
    dest: safe_cosafe[rotated] lpd_fv_regex[where fv=fv] split: if_splits)
qed (auto split: nat.splits)

lemma safe_rpd_fv_regex_le: safe_regex Past Strict  $r \implies s \in \text{rpd test } i \ r \implies \text{fv\_regex fv } r \subseteq \text{fv\_regex}$ 
 $\text{fv } s$ 
  by (induct r) (auto simp: TimesL_def split: if_splits)

lemma safe_rpd_fv_regex: safe_regex Past Strict  $r \implies s \in \text{rpd test } i \ r \implies \text{fv\_regex fv } s = \text{fv\_regex fv}$ 
 $r$ 
  by (simp add: eq_iff_rpd_fv_regex safe_rpd_fv_regex_le)

lemma cosafe_rpd: safe_regex Past Lax  $r \implies s \in \text{rpd test } i \ r \implies \text{safe\_regex Past Lax } s$ 
proof (induct r arbitrary: s)
  case (Plus r1 r2)
  from Plus(3,4) show ?case
  by (auto elim: Plus(1,2))
next
  case (Times r1 r2)
  from Times(3,4) show ?case
  by (auto simp: TimesL_def elim: Times(1,2) split: if_splits)
qed (auto simp: TimesL_def split: nat.splits)

lemma safe_rpd: ( $\forall x \in \text{atms } r. \text{safe Strict } x \longrightarrow \text{safe Lax } x$ )  $\implies$ 
  safe_regex Past Strict  $r \implies s \in \text{rpd test } i \ r \implies \text{safe\_regex Past Strict } s$ 
proof (induct r arbitrary: s)
  case (Plus r1 r2)
  from Plus(3,4,5) show ?case
  by (auto elim: Plus(1,2) simp: ball_Un)
next
  case (Times r1 r2)
  from Times(3,4,5) show ?case
  by (force simp: TimesL_def ball_Un elim: Times(1,2) cosafe_rpd dest: rpd_fv_regex split: if_splits)
next
  case (Star r)
  from Star(2,3,4) show ?case
  by (force simp: TimesL_def elim: Star(1) cosafe_rpd
    dest: safe_cosafe[rotated] rpd_fv_regex[where fv=fv] split: if_splits)
qed (auto split: nat.splits)

```

lemma *safe_regex_safe*: $(\bigwedge g r. \text{safe } g r \implies \text{safe } \text{Lax } r) \implies$
 $\text{safe_regex } m g r \implies x \in \text{atms } r \implies \text{safe } \text{Lax } x$
by (*induct* $m g r$ *rule*: *safe_regex.induct*) *auto*

lemma *safe_regex_map_regex*:
 $(\bigwedge g x. x \in \text{atms } r \implies \text{safe } g x \implies \text{safe } g (f x)) \implies (\bigwedge x. x \in \text{atms } r \implies \text{fv } (f x) = \text{fv } x) \implies$
 $\text{safe_regex } m g r \implies \text{safe_regex } m g (\text{map_regex } f r)$
by (*induct* $m g r$ *rule*: *safe_regex.induct*) (*auto simp*: *fv_regex_alt regex.set_map*)

end

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

lemma *safe_regex_mono*:
 $(\bigwedge g x. x \in \text{atms } r \implies \text{safe } g x \implies \text{safe}' g x) \implies$
 $\text{Regex.safe_regex } \text{fv } \text{safe } m g r \implies \text{Regex.safe_regex } \text{fv } \text{safe}' m g r$
by (*induct* $m g r$ *rule*: *safe_regex.induct*) *auto*

lemma *match_map_regex*: $\text{match } t (\text{map_regex } f r) = \text{match } (\lambda k z. t k (f z)) r$
by (*induct* r) *auto*

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

proof (*induction* r *arbitrary*: $i j$)

case (*Times* $r s$)

from *Times.prem*s **show** *?case*

by (*auto* $0 \ 4$ *simp*: *relcompp_apply intro*: *le_less_trans match_le less_Suc_eq_le*

dest: *Times.IH*[*THEN iffD1*, *rotated -1*] *Times.IH*[*THEN iffD2*, *rotated -1*] *match_le*)

next

case (*Star* r)

show *?case* **unfolding** *match.simp*s

proof (*rule iffI*)

assume $*$: $(\text{match } t r)^{**} i j$

then have $i \leq j$ **unfolding** *match.simp*s(5)[*symmetric*]

by (*rule match_le*)

with $*$ **show** $(\text{match } t' r)^{**} i j$ **using** *Star.prem*s

proof (*induction* $i j$ *rule*: *rtranclp.induct*)

case (*rtrancl_into_rtrancl* $a b c$)

from *rtrancl_into_rtrancl*(1,2,4,5) **show** *?case*

by (*intro rtranclp.rtrancl_into_rtrancl*[*OF rtrancl_into_rtrancl.IH*])

(*auto dest*!: *Star.IH*[*THEN iffD1*, *rotated -1*])

dest: *match_le match_rtranclp_le simp*: *less_Suc_eq_le*)

qed *simp*

next

assume $*$: $(\text{match } t' r)^{**} i j$

then have $i \leq j$ **unfolding** *match.simp*s(5)[*symmetric*]

by (*rule match_le*)

with $*$ **show** $(\text{match } t r)^{**} i j$ **using** *Star.prem*s

proof (*induction* $i j$ *rule*: *rtranclp.induct*)

case (*rtrancl_into_rtrancl* $a b c$)

from *rtrancl_into_rtrancl*(1,2,4,5) **show** *?case*

by (*intro rtranclp.rtrancl_into_rtrancl*[*OF rtrancl_into_rtrancl.IH*])

(*auto dest*!: *Star.IH*[*THEN iffD2*, *rotated -1*])

dest: *match_le match_rtranclp_le simp*: *less_Suc_eq_le*)

```

    qed simp
  qed
qed auto

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

```
instance by intro_classes
```

```
(auto simp: ccompare_enat_def split: if_splits intro!: comparator.intro)
```

```
end
```

```
context begin
```

4.1 Formulas and satisfiability

```
qualified type_synonym name = String.literal
```

```
qualified type_synonym event = (name  $\times$  event_data list)
```

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

```
qualified type_synonym prefix = (name  $\times$  event_data list) prefix
```

```
qualified type_synonym trace = (name  $\times$  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  $\Rightarrow$  trm  $\Rightarrow$  nat set where
```

```
  fvi_trm b (Var x) = (if  $b \leq x$  then  $\{x - b\}$  else  $\{\}$ )
```

```
| fvi_trm b (Const _) =  $\{\}$ 
```

```
| fvi_trm b (Plus x y) = fvi_trm b x  $\cup$  fvi_trm b y
```

```
| fvi_trm b (Minus x y) = fvi_trm b x  $\cup$  fvi_trm b y
```

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

```
| fvi_trm b (Mult x y) = fvi_trm b x  $\cup$  fvi_trm b y
```

```
| fvi_trm b (Div x y) = fvi_trm b x  $\cup$  fvi_trm b y
```

```
| fvi_trm b (Mod x y) = fvi_trm b x  $\cup$  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  $\equiv$  fvi_trm 0
```

```
qualified primrec eval_trm :: env  $\Rightarrow$  trm  $\Rightarrow$  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 $eval_trm_fv_cong$: $\forall x \in fv_trm\ t. v ! x = v' ! x \implies eval_trm\ v\ t = eval_trm\ v'\ t$
by ($induction\ t$) $simp_all$

qualified datatype $agg_type = Agg_Cnt | Agg_Min | Agg_Max | Agg_Sum | Agg_Avg | Agg_Med$
qualified type synonym $agg_op = agg_type \times event_data$

definition $flatten_multiset :: (event_data \times enat)\ set \Rightarrow event_data\ list$ **where**
 $flatten_multiset\ M = concat\ (map\ (\lambda(x, c). replicate\ (the_enat\ c)\ x)\ (csorted_list_of_set\ M))$

fun $eval_agg_op :: agg_op \Rightarrow (event_data \times enat)\ set \Rightarrow event_data$ **where**
 $eval_agg_op\ (Agg_Cnt, y0)\ M = EInt\ (integer_of_int\ (length\ (flatten_multiset\ M)))$
| $eval_agg_op\ (Agg_Min, y0)\ M = (case\ flatten_multiset\ M\ of$
 $\quad [] \Rightarrow y0$
 $\quad | x \# xs \Rightarrow foldl\ min\ x\ xs)$
| $eval_agg_op\ (Agg_Max, y0)\ M = (case\ flatten_multiset\ M\ of$
 $\quad [] \Rightarrow y0$
 $\quad | x \# xs \Rightarrow foldl\ max\ x\ xs)$
| $eval_agg_op\ (Agg_Sum, y0)\ M = foldl\ plus\ y0\ (flatten_multiset\ M)$
| $eval_agg_op\ (Agg_Avg, y0)\ M = EFloat\ (let\ xs = flatten_multiset\ M\ in\ case\ xs\ of$
 $\quad [] \Rightarrow 0$
 $\quad | _ \Rightarrow double_of_event_data\ (foldl\ plus\ (EInt\ 0)\ xs)\ / double_of_int\ (length\ xs))$
| $eval_agg_op\ (Agg_Med, y0)\ M = EFloat\ (let\ xs = flatten_multiset\ M; u = length\ xs\ in$
 $\quad if\ u = 0\ then\ 0\ else$
 $\quad \quad let\ u' = u\ div\ 2\ in$
 $\quad \quad if\ even\ u\ then$
 $\quad \quad \quad (double_of_event_data\ (xs ! (u' - 1)) + double_of_event_data\ (xs ! u')) / double_of_int\ 2$
 $\quad \quad else\ double_of_event_data\ (xs ! u'))$

qualified datatype ($discs_sels$) $formula = Pred\ name\ trm\ list$
| $Let\ name\ formula\ formula$
| $Eq\ trm\ trm | Less\ trm\ trm | LessEq\ trm\ trm$
| $Neg\ formula | Or\ formula\ formula | And\ formula\ formula | Ands\ formula\ list | Exists\ formula$
| $Agg\ nat\ agg_op\ nat\ trm\ formula$
| $Prev\ \mathcal{I}\ formula | Next\ \mathcal{I}\ formula$
| $Since\ formula\ \mathcal{I}\ formula | Until\ formula\ \mathcal{I}\ formula$
| $MatchF\ \mathcal{I}\ formula\ Regex.regex | MatchP\ \mathcal{I}\ formula\ Regex.regex$

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

qualified definition $TT \equiv Neg\ FF$

qualified fun $fvi :: nat \Rightarrow formula \Rightarrow nat\ set$ **where**
 $fvi\ b\ (Pred\ r\ ts) = (\bigcup t \in set\ ts. fvi_trm\ b\ t)$
| $fvi\ b\ (Let\ p\ \varphi\ \psi) = fvi\ b\ \psi$
| $fvi\ b\ (Eq\ t1\ t2) = fvi_trm\ b\ t1 \cup fvi_trm\ b\ t2$
| $fvi\ b\ (Less\ t1\ t2) = fvi_trm\ b\ t1 \cup fvi_trm\ b\ t2$
| $fvi\ b\ (LessEq\ t1\ t2) = fvi_trm\ b\ t1 \cup fvi_trm\ b\ t2$
| $fvi\ b\ (Neg\ \varphi) = fvi\ b\ \varphi$
| $fvi\ b\ (Or\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
| $fvi\ b\ (And\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
| $fvi\ b\ (Ands\ \varphi s) = (let\ xs = map\ (fvi\ b)\ \varphi s\ in\ \bigcup x \in set\ xs. x)$
| $fvi\ b\ (Exists\ \varphi) = fvi\ (Suc\ b)\ \varphi$
| $fvi\ b\ (Agg\ y\ \omega\ b'\ f\ \varphi) = fvi\ (b + b')\ \varphi \cup fvi_trm\ (b + b')\ f \cup (if\ b \leq y\ then\ \{y - b\}\ else\ \{\})$

```

| fvi b (Prev I φ) = fvi b φ
| fvi b (Next I φ) = fvi b φ
| fvi b (Since φ I ψ) = fvi b φ ∪ fvi b ψ
| fvi b (Until φ I ψ) = fvi b φ ∪ fvi b ψ
| fvi b (MatchF I r) = Regex.fv_regex (fvi b) r
| fvi b (MatchP I r) = Regex.fv_regex (fvi b) r

```

abbreviation $fv \equiv fvi\ 0$

abbreviation $fv_regex \equiv Regex.fv_regex\ fv$

lemma $fv_abbrevs[simp]: fv\ TT = \{\} fv\ FF = \{\}$
unfolding $TT_def\ FF_def$ **by** *auto*

lemma $fv_subset_Ands: \varphi \in set\ \varphi s \implies fv\ \varphi \subseteq fv\ (Ands\ \varphi s)$
by *auto*

lemma $finite_fvi_trm[simp]: finite\ (fvi_trm\ b\ t)$
by (*induction t simp_all*)

lemma $finite_fvi[simp]: finite\ (fvi\ b\ \varphi)$
by (*induction φ arbitrary: b simp_all*)

lemma $fvi_trm_plus: x \in fvi_trm\ (b + c)\ t \longleftrightarrow x + c \in fvi_trm\ b\ t$
by (*induction t auto*)

lemma $fvi_trm_iff_fv_trm: x \in fvi_trm\ b\ t \longleftrightarrow x + b \in fv_trm\ t$
using $fvi_trm_plus[where\ b=0\ and\ c=b]$ **by** *simp_all*

lemma $fvi_plus: x \in fvi\ (b + c)\ \varphi \longleftrightarrow x + c \in fvi\ b\ \varphi$

proof (*induction φ arbitrary: b rule: formula.induct*)

case (*Exists φ*)

then show *?case* **by** *force*

next

case (*Agg y ω b' f φ*)

have $*$: $b + c + b' = b + b' + c$ **by** *simp*

from *Agg* **show** *?case* **by** (*force simp: * fvi_trm_plus*)

qed (*auto simp add: fvi_trm_plus fv_regex_commute[where g = λx. x + c]*)

lemma $fvi_Suc: x \in fvi\ (Suc\ b)\ \varphi \longleftrightarrow Suc\ x \in fvi\ b\ \varphi$
using $fvi_plus[where\ c=1]$ **by** *simp*

lemma $fvi_plus_bound:$

assumes $\forall i \in fvi\ (b + c)\ \varphi. i < n$

shows $\forall i \in fvi\ b\ \varphi. i < c + n$

proof

fix i

assume $i \in fvi\ b\ \varphi$

show $i < c + n$

proof (*cases i < c*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

then obtain i' **where** $i = i' + c$

using $nat_le_iff_add$ **by** (*auto simp: not_less*)

with *assms* $\langle i \in fvi\ b\ \varphi \rangle$ **show** *?thesis* **by** (*simp add: fvi_plus*)

qed

qed

lemma *fv_i_Suc_bound*:
assumes $\forall i \in \text{fv } i \text{ (Suc } b) \varphi. i < n$
shows $\forall i \in \text{fv } i \text{ } b \varphi. i < \text{Suc } n$
using *assms fv_plus_bound* [where $c=1$] **by** *simp*

lemma *fv_i_iff_fv*: $x \in \text{fv } i \text{ } b \varphi \longleftrightarrow x + b \in \text{fv } \varphi$
using *fv_plus* [where $b=0$ and $c=b$] **by** *simp_all*

qualified definition *nfv* :: *formula* \Rightarrow *nat* **where**
 $\text{nfv } \varphi = \text{Max } (\text{insert } 0 \text{ (Suc 'fv } \varphi))$

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*]:
 $\text{nfv } (\text{Let } p \text{ } \varphi \text{ } \psi) = \text{nfv } \psi$
 $\text{nfv } (\text{Neg } \varphi) = \text{nfv } \varphi$
 $\text{nfv } (\text{Or } \varphi \text{ } \psi) = \text{max } (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{And } \varphi \text{ } \psi) = \text{max } (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{Prev } I \text{ } \varphi) = \text{nfv } \varphi$
 $\text{nfv } (\text{Next } I \text{ } \varphi) = \text{nfv } \varphi$
 $\text{nfv } (\text{Since } \varphi \text{ } I \text{ } \psi) = \text{max } (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{Until } \varphi \text{ } I \text{ } \psi) = \text{max } (\text{nfv } \varphi) (\text{nfv } \psi)$
 $\text{nfv } (\text{MatchP } I \text{ } r) = \text{Regex.nfv_regex } \text{fv } r$
 $\text{nfv } (\text{MatchF } I \text{ } r) = \text{Regex.nfv_regex } \text{fv } r$
 $\text{nfv_regex } (\text{Regex.Skip } n) = 0$
 $\text{nfv_regex } (\text{Regex.Test } \varphi) = \text{Max } (\text{insert } 0 \text{ (Suc 'fv } \varphi))$
 $\text{nfv_regex } (\text{Regex.Plus } r \text{ } s) = \text{max } (\text{nfv_regex } r) (\text{nfv_regex } s)$
 $\text{nfv_regex } (\text{Regex.Times } r \text{ } s) = \text{max } (\text{nfv_regex } r) (\text{nfv_regex } s)$
 $\text{nfv_regex } (\text{Regex.Star } r) = \text{nfv_regex } r$
unfolding *nfv_def* *Regex.nfv_regex_def* **by** (*simp_all* *add: image_Un Max_Un* [*symmetric*])

lemma *nfv_Ands* [*simp*]: $\text{nfv } (\text{Ands } l) = \text{Max } (\text{insert } 0 \text{ (nfv 'set } l))$
proof (*induction* *l*)
case *Nil*
then show *?case* **unfolding** *nfv_def* **by** *simp*
next
case (*Cons* *a l*)
have $\text{fv } (\text{Ands } (a \# l)) = \text{fv } a \cup \text{fv } (\text{Ands } l)$ **by** *simp*
then have $\text{nfv } (\text{Ands } (a \# l)) = \text{max } (\text{nfv } a) (\text{nfv } (\text{Ands } l))$
unfolding *nfv_def*
by (*auto simp: image_Un Max.union* [*symmetric*])
with *Cons.IH* **show** *?case*
by (*cases* *l*) *auto*
qed

lemma *fv_i_less_nfv*: $\forall i \in \text{fv } \varphi. i < \text{nfv } \varphi$
unfolding *nfv_def*
by (*auto simp add: Max_gr_iff* *intro: max.strict_coboundedI2*)

lemma *fv_i_less_nfv_regex*: $\forall i \in \text{fv_regex } \varphi. i < \text{nfv_regex } \varphi$
unfolding *Regex.nfv_regex_def*
by (*auto simp add: Max_gr_iff* *intro: max.strict_coboundedI2*)

4.1.2 Future reach

qualified fun `future_bounded` :: `formula` \Rightarrow `bool` **where**

```

future_bounded (Pred _ _) = True
| future_bounded (Let p  $\varphi$   $\psi$ ) = (future_bounded  $\varphi$   $\wedge$  future_bounded  $\psi$ )
| future_bounded (Eq _ _) = True
| future_bounded (Less _ _) = True
| future_bounded (LessEq _ _) = True
| future_bounded (Neg  $\varphi$ ) = future_bounded  $\varphi$ 
| future_bounded (Or  $\varphi$   $\psi$ ) = (future_bounded  $\varphi$   $\wedge$  future_bounded  $\psi$ )
| future_bounded (And  $\varphi$   $\psi$ ) = (future_bounded  $\varphi$   $\wedge$  future_bounded  $\psi$ )
| future_bounded (Ands l) = list_all future_bounded l
| future_bounded (Exists  $\varphi$ ) = future_bounded  $\varphi$ 
| future_bounded (Agg y  $\omega$  b f  $\varphi$ ) = future_bounded  $\varphi$ 
| future_bounded (Prev I  $\varphi$ ) = future_bounded  $\varphi$ 
| future_bounded (Next I  $\varphi$ ) = future_bounded  $\varphi$ 
| future_bounded (Since  $\varphi$  I  $\psi$ ) = (future_bounded  $\varphi$   $\wedge$  future_bounded  $\psi$ )
| future_bounded (Until  $\varphi$  I  $\psi$ ) = (future_bounded  $\varphi$   $\wedge$  future_bounded  $\psi$   $\wedge$  right I  $\neq$   $\infty$ )
| future_bounded (MatchP I r) = Regex.pred_regex future_bounded r
| future_bounded (MatchF I r) = (Regex.pred_regex future_bounded r  $\wedge$  right I  $\neq$   $\infty$ )

```

4.1.3 Semantics

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

qualified fun `sat` :: `trace` \Rightarrow (`name` \rightarrow `nat` \Rightarrow `event_data list set`) \Rightarrow `env` \Rightarrow `nat` \Rightarrow `formula` \Rightarrow `bool`
where

```

sat  $\sigma$  V v i (Pred r ts) = (case V r of
  None  $\Rightarrow$  (r, map (eval_trm v) ts)  $\in$   $\Gamma$   $\sigma$  i
  | Some X  $\Rightarrow$  map (eval_trm v) ts  $\in$  X i)
| sat  $\sigma$  V v i (Let p  $\varphi$   $\psi$ ) =
  sat  $\sigma$  (V (p  $\mapsto$   $\lambda$ i. {v. length v = nfv  $\varphi$   $\wedge$  sat  $\sigma$  V v i  $\varphi$ })) v i  $\psi$ 
| sat  $\sigma$  V v i (Eq t1 t2) = (eval_trm v t1 = eval_trm v t2)
| sat  $\sigma$  V v i (Less t1 t2) = (eval_trm v t1 < eval_trm v t2)
| sat  $\sigma$  V v i (LessEq t1 t2) = (eval_trm v t1  $\leq$  eval_trm v t2)
| sat  $\sigma$  V v i (Neg  $\varphi$ ) = ( $\neg$  sat  $\sigma$  V v i  $\varphi$ )
| sat  $\sigma$  V v i (Or  $\varphi$   $\psi$ ) = (sat  $\sigma$  V v i  $\varphi$   $\vee$  sat  $\sigma$  V v i  $\psi$ )
| sat  $\sigma$  V v i (And  $\varphi$   $\psi$ ) = (sat  $\sigma$  V v i  $\varphi$   $\wedge$  sat  $\sigma$  V v i  $\psi$ )
| sat  $\sigma$  V v i (Ands l) = ( $\forall$   $\varphi$   $\in$  set l. sat  $\sigma$  V v i  $\varphi$ )
| sat  $\sigma$  V v i (Exists  $\varphi$ ) = ( $\exists$  z. sat  $\sigma$  V (z # v) i  $\varphi$ )
| sat  $\sigma$  V v i (Agg y  $\omega$  b f  $\varphi$ ) =
  (let M = {(x, ecard Zs) | x Zs. Zs = {zs. length zs = b  $\wedge$  sat  $\sigma$  V (zs @ v) i  $\varphi$   $\wedge$  eval_trm (zs @ v) f
= x}  $\wedge$  Zs  $\neq$  {}}
  in (M = {})  $\rightarrow$  fv  $\varphi$   $\subseteq$  {0..}  $\wedge$  v ! y = eval_agg_op  $\omega$  M)
| sat  $\sigma$  V v i (Prev I  $\varphi$ ) = (case i of 0  $\Rightarrow$  False | Suc j  $\Rightarrow$  mem ( $\tau$   $\sigma$  i -  $\tau$   $\sigma$  j) I  $\wedge$  sat  $\sigma$  V v j  $\varphi$ )
| sat  $\sigma$  V v i (Next I  $\varphi$ ) = (mem ( $\tau$   $\sigma$  (Suc i) -  $\tau$   $\sigma$  i) I  $\wedge$  sat  $\sigma$  V v (Suc i)  $\varphi$ )
| sat  $\sigma$  V v i (Since  $\varphi$  I  $\psi$ ) = ( $\exists$  j  $\leq$  i. mem ( $\tau$   $\sigma$  i -  $\tau$   $\sigma$  j) I  $\wedge$  sat  $\sigma$  V v j  $\psi$   $\wedge$  ( $\forall$  k  $\in$  {j <.. i}. sat  $\sigma$  V
v k  $\varphi$ ))
| sat  $\sigma$  V v i (Until  $\varphi$  I  $\psi$ ) = ( $\exists$  j  $\geq$  i. mem ( $\tau$   $\sigma$  j -  $\tau$   $\sigma$  i) I  $\wedge$  sat  $\sigma$  V v j  $\psi$   $\wedge$  ( $\forall$  k  $\in$  {i ..< j}. sat  $\sigma$  V
v k  $\varphi$ ))
| sat  $\sigma$  V v i (MatchP I r) = ( $\exists$  j  $\leq$  i. mem ( $\tau$   $\sigma$  i -  $\tau$   $\sigma$  j) I  $\wedge$  Regex.match (sat  $\sigma$  V v) r j i)
| sat  $\sigma$  V v i (MatchF I r) = ( $\exists$  j  $\geq$  i. mem ( $\tau$   $\sigma$  j -  $\tau$   $\sigma$  i) I  $\wedge$  Regex.match (sat  $\sigma$  V v) r i j)

```

lemma `sat_abbrevs`[simp]:

```

sat  $\sigma$  V v i TT  $\neg$  sat  $\sigma$  V v i FF
unfolding TT_def FF_def by auto

```

lemma `sat_Ands`: sat σ V v i (Ands l) \longleftrightarrow (\forall φ \in set l. sat σ V v i φ)

by (simp add: list_all_iff)

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

proof (rule iffI; (elim disjE conjE)?)
assume ?L
then obtain j where $j: i \leq j \text{ mem } (\tau \sigma \ j - \tau \sigma \ i) \ I \text{ sat } \sigma \ V \ v \ j \ \psi \ \forall k \in \{i ..< j\}. \text{sat } \sigma \ V \ v \ k \ \varphi$
by auto
then show ?R
proof (cases i = j)
case False
with $j(1,2)$ **have** $\Delta \sigma \ (i + 1) \leq \text{right } I$
by (auto elim: order_trans[rotated] simp: diff_le_mono)
moreover from False $j(1,4)$ **have** $\text{sat } \sigma \ V \ v \ i \ \varphi$ **by** auto
moreover from False j **have** $\text{sat } \sigma \ V \ v \ (i + 1) \ (\text{Until } \varphi \ (\text{subtract } (\Delta \sigma \ (i + 1)) \ I) \ \psi)$
by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
ultimately show ?thesis **by** blast
qed simp
next
assume $\Delta: \Delta \sigma \ (i + 1) \leq \text{right } I$ **and now:** $\text{sat } \sigma \ V \ v \ i \ \varphi$ **and**
next: $\text{sat } \sigma \ V \ v \ (i + 1) \ (\text{Until } \varphi \ (\text{subtract } (\Delta \sigma \ (i + 1)) \ I) \ \psi)$
from next obtain j where $j: i + 1 \leq j \text{ mem } (\tau \sigma \ j - \tau \sigma \ (i + 1)) \ ((\text{subtract } (\Delta \sigma \ (i + 1)) \ I))$
 $\text{sat } \sigma \ V \ v \ j \ \psi \ \forall k \in \{i + 1 ..< j\}. \text{sat } \sigma \ V \ v \ k \ \varphi$
by auto
from $\Delta \ j(1,2)$ **have** $\text{mem } (\tau \sigma \ j - \tau \sigma \ i) \ I$
by (cases right I) (auto simp: le_diff_conv2)
with now $j(1,3,4)$ **show** ?L **by** (auto simp: le_eq_less_or_eq[of i] intro!: exI[of _ j])
qed auto

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

proof (rule iffI; (elim disjE conjE)?)
assume ?L
then obtain j where $j: j \leq i \text{ mem } (\tau \sigma \ i - \tau \sigma \ j) \ I \text{ sat } \sigma \ V \ v \ j \ \psi \ \forall k \in \{j <.. i\}. \text{sat } \sigma \ V \ v \ k \ \varphi$
by auto
then show ?R
proof (cases i = j)
case False
with $j(1)$ **obtain k where** [simp]: $i = k + 1$
by (cases i) auto
with $j(1,2)$ False **have** $\Delta \sigma \ i \leq \text{right } I$
by (auto elim: order_trans[rotated] simp: diff_le_mono2 le_Suc_eq)
moreover from False $j(1,4)$ **have** $\text{sat } \sigma \ V \ v \ i \ \varphi$ **by** auto
moreover from False j **have** $\text{sat } \sigma \ V \ v \ (i - 1) \ (\text{Since } \varphi \ (\text{subtract } (\Delta \sigma \ i) \ I) \ \psi)$
by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
ultimately show ?thesis **by** auto
qed simp
next
assume $i: 0 < i$ **and** $\Delta: \Delta \sigma \ i \leq \text{right } I$ **and now:** $\text{sat } \sigma \ V \ v \ i \ \varphi$ **and**
prev: $\text{sat } \sigma \ V \ v \ (i - 1) \ (\text{Since } \varphi \ (\text{subtract } (\Delta \sigma \ i) \ I) \ \psi)$
from prev obtain j where $j: j \leq i - 1 \text{ mem } (\tau \sigma \ (i - 1) - \tau \sigma \ j) \ ((\text{subtract } (\Delta \sigma \ i) \ I))$
 $\text{sat } \sigma \ V \ v \ j \ \psi \ \forall k \in \{j <.. i - 1\}. \text{sat } \sigma \ V \ v \ k \ \varphi$
by auto
from $\Delta \ i \ j(1,2)$ **have** $\text{mem } (\tau \sigma \ i - \tau \sigma \ j) \ I$
by (cases right I) (auto simp: le_diff_conv2)

with $now\ i\ j(1,3,4)$ **show** $?L$ **by** $(auto\ simp:\ le_Suc_eq\ gr0_conv_Suc\ intro!:\ exI[of_j])$
qed $auto$

lemma $sat_MatchF_rec:$ $sat\ \sigma\ V\ v\ i\ (MatchF\ I\ r) \longleftrightarrow mem\ 0\ I \wedge Regex.eps\ (sat\ \sigma\ V\ v)\ i\ r \vee$
 $\Delta\ \sigma\ (i + 1) \leq right\ I \wedge (\exists s \in Regex.lpd\ (sat\ \sigma\ V\ v)\ i\ r.\ sat\ \sigma\ V\ v\ (i + 1)\ (MatchF\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ s))$
(is $?L \longleftrightarrow ?R1 \vee ?R2$
proof $(rule\ iffI;\ (elim\ disjE\ conjE\ bexE)?)$
assume $?L$
then obtain j **where** $j: j \geq i$ $mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$ **and** $Regex.match\ (sat\ \sigma\ V\ v)\ r\ i\ j$ **by** $auto$
then show $?R1 \vee ?R2$
proof $(cases\ i < j)$
case $True$
with $\langle Regex.match\ (sat\ \sigma\ V\ v)\ r\ i\ j \rangle$ $lpd_match[of\ i\ j\ sat\ \sigma\ V\ v\ r]$
obtain s **where** $s \in Regex.lpd\ (sat\ \sigma\ V\ v)\ i\ r$ $Regex.match\ (sat\ \sigma\ V\ v)\ s\ (i + 1)\ j$ **by** $auto$
with $True\ j$ **have** $?R2$
by $(cases\ right\ I)$
 $(auto\ simp:\ le_diff_conv\ le_diff_conv2\ intro!:\ exI[of_j]\ elim:\ le_trans[rotated])$
then show $?thesis$ **by** $blast$
qed $(auto\ simp:\ eps_match)$

next
assume $enat\ (\Delta\ \sigma\ (i + 1)) \leq right\ I$
moreover
fix s
assume $[simp]: s \in Regex.lpd\ (sat\ \sigma\ V\ v)\ i\ r$ **and** $sat\ \sigma\ V\ v\ (i + 1)\ (MatchF\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ s)$
then obtain j **where** $j > i$ $Regex.match\ (sat\ \sigma\ V\ v)\ s\ (i + 1)\ j$
 $mem\ (\tau\ \sigma\ j - \tau\ \sigma\ (Suc\ i))\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)$ **by** $(auto\ simp:\ Suc_le_eq)$
ultimately show $?L$
by $(cases\ right\ I)$
 $(auto\ simp:\ le_diff_conv\ lpd_match\ intro!:\ exI[of_j]\ beXI[of_s])$
qed $(auto\ simp:\ eps_match\ intro!:\ exI[of_i])$

lemma $sat_MatchP_rec:$ $sat\ \sigma\ V\ v\ i\ (MatchP\ I\ r) \longleftrightarrow mem\ 0\ I \wedge Regex.eps\ (sat\ \sigma\ V\ v)\ i\ r \vee$
 $i > 0 \wedge \Delta\ \sigma\ i \leq right\ I \wedge (\exists s \in Regex.rpd\ (sat\ \sigma\ V\ v)\ i\ r.\ sat\ \sigma\ V\ v\ (i - 1)\ (MatchP\ (subtract\ (\Delta\ \sigma\ i)\ I)\ s))$
(is $?L \longleftrightarrow ?R1 \vee ?R2$
proof $(rule\ iffI;\ (elim\ disjE\ conjE\ bexE)?)$
assume $?L$
then obtain j **where** $j: j \leq i$ $mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$ **and** $Regex.match\ (sat\ \sigma\ V\ v)\ r\ j\ i$ **by** $auto$
then show $?R1 \vee ?R2$
proof $(cases\ j < i)$
case $True$
with $\langle Regex.match\ (sat\ \sigma\ V\ v)\ r\ j\ i \rangle$ $rpd_match[of\ j\ i\ sat\ \sigma\ V\ v\ r]$
obtain s **where** $s \in Regex.rpd\ (sat\ \sigma\ V\ v)\ i\ r$ $Regex.match\ (sat\ \sigma\ V\ v)\ s\ j\ (i - 1)$ **by** $auto$
with $True\ j$ **have** $?R2$
by $(cases\ right\ I)$
 $(auto\ simp:\ le_diff_conv\ le_diff_conv2\ intro!:\ exI[of_j]\ elim:\ le_trans)$
then show $?thesis$ **by** $blast$
qed $(auto\ simp:\ eps_match)$

next
assume $enat\ (\Delta\ \sigma\ i) \leq right\ I$
moreover
fix s
assume $[simp]: s \in Regex.rpd\ (sat\ \sigma\ V\ v)\ i\ r$ **and** $sat\ \sigma\ V\ v\ (i - 1)\ (MatchP\ (subtract\ (\Delta\ \sigma\ i)\ I)\ s)$
 $i > 0$
then obtain j **where** $j < i$ $Regex.match\ (sat\ \sigma\ V\ v)\ s\ j\ (i - 1)$
 $mem\ (\tau\ \sigma\ (i - 1) - \tau\ \sigma\ j)\ (subtract\ (\Delta\ \sigma\ i)\ I)$ **by** $(auto\ simp:\ gr0_conv_Suc\ less_Suc_eq_le)$

ultimately show ?L
by (cases right I)
(auto simp: le_diff_conv rpd_match intro!: exI[of _ j] beXI[of _ s])
qed (auto simp: eps_match intro!: exI[of _ i])

lemma sat_Since_0: sat σ V v 0 (Since φ I ψ) \longleftrightarrow mem 0 I \wedge sat σ V v 0 ψ
by auto

lemma sat_MatchP_0: sat σ V v 0 (MatchP I r) \longleftrightarrow mem 0 I \wedge Regex.eps (sat σ V v) 0 r
by (auto simp: eps_match)

lemma sat_Since_point: sat σ V v i (Since φ I ψ) \implies
($\bigwedge j. j \leq i \implies$ mem ($\tau \sigma i - \tau \sigma j$) I \implies sat σ V v i (Since φ (point ($\tau \sigma i - \tau \sigma j$)) ψ)) \implies P)
 \implies P
by (auto intro: diff_le_self)

lemma sat_MatchP_point: sat σ V v i (MatchP I r) \implies
($\bigwedge j. j \leq i \implies$ mem ($\tau \sigma i - \tau \sigma j$) I \implies sat σ V v i (MatchP (point ($\tau \sigma i - \tau \sigma j$)) r)) \implies P)
 \implies P
by (auto intro: diff_le_self)

lemma sat_Since_pointD: sat σ V v i (Since φ (point t) ψ) \implies mem t I \implies sat σ V v i (Since φ I ψ)
by auto

lemma sat_MatchP_pointD: sat σ V v i (MatchP (point t) r) \implies mem t I \implies sat σ V v i (MatchP I r)
by auto

lemma sat_fv_cong: $\forall x \in fv \varphi. v!x = v!x \implies$ sat σ V v i $\varphi =$ sat σ V v' i φ
proof (induct φ arbitrary: V v v' i rule: formula.induct)
case (Pred n ts)
show ?case **by** (simp cong: map_cong eval_trm_fv_cong[OF Pred[simplified, THEN bspec]] split: option.splits)
next
case (Let p b φ ψ)
then show ?case
by auto
next
case (Eq x1 x2)
then show ?case **unfolding** fvi.simps sat.simps **by** (metis UnCI eval_trm_fv_cong)
next
case (Less x1 x2)
then show ?case **unfolding** fvi.simps sat.simps **by** (metis UnCI eval_trm_fv_cong)
next
case (LessEq x1 x2)
then show ?case **unfolding** fvi.simps sat.simps **by** (metis UnCI eval_trm_fv_cong)
next
case (Ands l)
have $\bigwedge \varphi. \varphi \in set l \implies$ sat σ V v i $\varphi =$ sat σ V v' i φ
proof -
fix φ **assume** $\varphi \in set l$
then have $fv \varphi \subseteq fv (Ands l)$ **using** fv_subset_Ands **by** blast
then have $\forall x \in fv \varphi. v!x = v!x$ **using** Ands.prem1s **by** blast
then show sat σ V v i $\varphi =$ sat σ V v' i φ **using** Ands.hyps $\langle \varphi \in set l \rangle$ **by** blast
qed
then show ?case **using** sat_Ands **by** blast
next
case (Exists φ)

```

    then show ?case unfolding sat.simps by (intro iff_exI) (simp add: fvi_Suc nth_Cons')
next
case (Agg y ω b f φ)
have v ! y = v' ! y
  using Agg.premis by simp
moreover have sat σ V (zs @ v) i φ = sat σ V (zs @ v') i φ if length zs = b for zs
  using that Agg.premis by (simp add: Agg.hyps[where v=zs @ v and v'=zs @ v]
    nth_append fvi_iff_fv(1)[where b=b])
moreover have eval_trm (zs @ v) f = eval_trm (zs @ v') f if length zs = b for zs
  using that Agg.premis by (auto intro!: eval_trm_fv_cong[where v=zs @ v and v'=zs @ v]
    simp: nth_append fvi_iff_fv(1)[where b=b] fvi_trm_iff_fv_trm[where b=length zs])
ultimately show ?case
  by (simp cong: conj_cong)
next
case (MatchF I r)
then have Regex.match (sat σ V v) r = Regex.match (sat σ V v') r
  by (intro match_fv_cong) (auto simp: fv_regex_alt)
then show ?case
  by auto
next
case (MatchP I r)
then have Regex.match (sat σ V v) r = Regex.match (sat σ V v') r
  by (intro match_fv_cong) (auto simp: fv_regex_alt)
then show ?case
  by auto
qed (auto 10 0 split: nat.splits intro!: iff_exI)

```

lemma match_fv_cong:

$\forall x \in \text{fv_regex } r. v!x = v'!x \implies \text{Regex.match (sat } \sigma V v) r = \text{Regex.match (sat } \sigma V v') r$
 by (rule match_fv_cong, rule sat_fv_cong) (auto simp: fv_regex_alt)

lemma eps_fv_cong:

$\forall x \in \text{fv_regex } r. v!x = v'!x \implies \text{Regex.eps (sat } \sigma V v) i r = \text{Regex.eps (sat } \sigma V v') i r$
 unfolding eps_match by (erule match_fv_cong[THEN fun_cong, THEN fun_cong])

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 _ α β) = (past_only α ∧ past_only β)
| past_only (Neg ψ) = past_only ψ
| past_only (Or α β) = (past_only α ∧ past_only β)
| past_only (And α β) = (past_only α ∧ past_only β)
| past_only (Ands l) = ( $\forall \alpha \in \text{set } l. \text{past\_only } \alpha$ )
| past_only (Exists ψ) = past_only ψ
| past_only (Agg _ _ _ _ ψ) = past_only ψ
| past_only (Prev _ ψ) = past_only ψ
| past_only (Next _ _) = False
| past_only (Since α _ β) = (past_only α ∧ past_only β)
| past_only (Until α _ β) = False
| past_only (MatchP _ r) = Regex.pred_regex past_only r
| past_only (MatchF _ _) = False

```

lemma past_only_sat:

assumes prefix_of π σ prefix_of π σ'

```

shows  $i < \text{plen } \pi \implies \text{dom } V = \text{dom } V' \implies$ 
   $(\bigwedge p \ i. \ p \in \text{dom } V \implies i < \text{plen } \pi \implies \text{the } (V \ p) \ i = \text{the } (V' \ p) \ i) \implies$ 
   $\text{past\_only } \varphi \implies \text{sat } \sigma \ V \ v \ i \ \varphi = \text{sat } \sigma' \ V' \ v \ i \ \varphi$ 
proof (induction  $\varphi$  arbitrary:  $V \ V' \ v \ i$ )
case ( $\text{Pred } e \ ts$ )
show  $?case$  proof ( $\text{cases } V \ e$ )
  case  $\text{None}$ 
    then have  $V' \ e = \text{None}$  using  $\langle \text{dom } V = \text{dom } V' \rangle$  by  $\text{auto}$ 
    with  $\text{None } \Gamma\_prefix\_conv[OF \ \text{assms}(1,2) \ \text{Pred}(1)]$  show  $?thesis$  by  $\text{simp}$ 
  next
    case ( $\text{Some } a$ )
    moreover obtain  $a'$  where  $V' \ e = \text{Some } a'$  using  $\text{Some } \langle \text{dom } V = \text{dom } V' \rangle$  by  $\text{auto}$ 
    moreover have  $\text{the } (V \ e) \ i = \text{the } (V' \ e) \ i$ 
      using  $\text{Some } \text{Pred}(1,3)$  by ( $\text{fastforce intro: domI}$ )
    ultimately show  $?thesis$  by  $\text{simp}$ 
  qed
next
case ( $\text{Let } p \ \varphi \ \psi$ )
let  $?V = \lambda V \ \sigma. \ (V(p \mapsto \lambda i. \ \{v. \ \text{length } v = \text{nfv } \varphi \wedge \text{sat } \sigma \ V \ v \ i \ \varphi\}))$ 
show  $?case$  unfolding  $\text{sat.simps}$  proof ( $\text{rule Let.IH}(2)$ )
  show  $i < \text{plen } \pi$  by  $\text{fact}$ 
  from  $\text{Let.prem}s$  show  $\text{past\_only } \psi$  by  $\text{simp}$ 
  from  $\text{Let.prem}s$  show  $\text{dom } (?V \ V \ \sigma) = \text{dom } (?V \ V' \ \sigma')$ 
    by ( $\text{simp del: fun\_upd\_apply}$ )
  next
    fix  $p' \ i$ 
    assume  $*$ :  $p' \in \text{dom } (?V \ V \ \sigma) \ i < \text{plen } \pi$ 
    show  $\text{the } (?V \ V \ \sigma \ p') \ i = \text{the } (?V \ V' \ \sigma' \ p') \ i$  proof ( $\text{cases } p' = p$ )
      case  $\text{True}$ 
        with  $\text{Let } \langle i < \text{plen } \pi \rangle$  show  $?thesis$  by  $\text{auto}$ 
      next
        case  $\text{False}$ 
        with  $*$  show  $?thesis$  by ( $\text{auto intro!: Let.prem}s(3)$ )
      qed
    qed
  next
    case ( $\text{Ands } l$ )
    with  $\Gamma\_prefix\_conv[OF \ \text{assms}]$  show  $?case$  by  $\text{simp}$ 
  next
    case ( $\text{Prev } I \ \varphi$ )
    with  $\tau\_prefix\_conv[OF \ \text{assms}]$  show  $?case$  by ( $\text{simp split: nat.split}$ )
  next
    case ( $\text{Since } \varphi 1 \ I \ \varphi 2$ )
    with  $\tau\_prefix\_conv[OF \ \text{assms}]$  show  $?case$  by  $\text{auto}$ 
  next
    case ( $\text{MatchP } I \ r$ )
    then have  $\text{Regex.match } (\text{sat } \sigma \ V \ v) \ r \ a \ b = \text{Regex.match } (\text{sat } \sigma' \ V' \ v) \ r \ a \ b$  if  $b < \text{plen } \pi$  for  $a \ b$ 
      using  $\text{that}$  by ( $\text{intro } \text{Regex.match\_cong\_strong}$ ) ( $\text{auto simp: regex.pred\_set}$ )
    with  $\tau\_prefix\_conv[OF \ \text{assms}] \ \text{MatchP}(2)$  show  $?case$  by  $\text{auto}$ 
  qed  $\text{auto}$ 

```

4.3 Safe formulas

fun $\text{remove_neg} :: \text{formula} \Rightarrow \text{formula}$ **where**

$\text{remove_neg } (\text{Neg } \varphi) = \varphi$
 $|\ \text{remove_neg } \varphi = \varphi$

lemma $\text{fvi_remove_neg}[simp]$: $\text{fvi } b \ (\text{remove_neg } \varphi) = \text{fvi } b \ \varphi$

by (cases φ) simp_all

lemma partition_cong[fundef_cong]:

$xs = ys \implies (\bigwedge x. x \in \text{set } xs \implies f x = g x) \implies \text{partition } f xs = \text{partition } g ys$

by (induction xs arbitrary: ys) auto

lemma size_remove_neg[termination_simp]: $\text{size } (\text{remove_neg } \varphi) \leq \text{size } \varphi$

by (cases φ) simp_all

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 φ = (case φ 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 φ ψ) = ($\{0..<\text{nfv } \varphi\} \subseteq \text{fv } \varphi \wedge \text{safe_formula } \varphi \wedge \text{safe_formula } \psi$)

| safe_formula (Neg φ) = ($\text{fv } \varphi = \{\}$) \wedge safe_formula φ)

| safe_formula (Or φ ψ) = ($\text{fv } \psi = \text{fv } \varphi \wedge \text{safe_formula } \varphi \wedge \text{safe_formula } \psi$)

| safe_formula (And φ ψ) = (safe_formula φ \wedge

(safe_assignment (fv φ) $\psi \vee \text{safe_formula } \psi \vee$

$\text{fv } \psi \subseteq \text{fv } \varphi \wedge (\text{is_constraint } \psi \vee (\text{case } \psi \text{ of Neg } \psi' \Rightarrow \text{safe_formula } \psi' \mid _ \Rightarrow \text{False}))))$)

| safe_formula (Ands l) = (let (pos, neg) = partition safe_formula l in pos \neq [] \wedge

list_all safe_formula (map remove_neg neg) $\wedge \bigcup (\text{set } (\text{map fv neg})) \subseteq \bigcup (\text{set } (\text{map fv pos})))$)

| safe_formula (Exists φ) = (safe_formula φ)

| safe_formula (Agg y ω b f φ) = (safe_formula $\varphi \wedge y + b \notin \text{fv } \varphi \wedge \{0..<b\} \subseteq \text{fv } \varphi \wedge \text{fv_trm } f \subseteq \text{fv } \varphi$)

| safe_formula (Prev I φ) = (safe_formula φ)

| safe_formula (Next I φ) = (safe_formula φ)

| safe_formula (Since φ I ψ) = ($\text{fv } \varphi \subseteq \text{fv } \psi \wedge$

(safe_formula $\varphi \vee (\text{case } \varphi \text{ of Neg } \varphi' \Rightarrow \text{safe_formula } \varphi' \mid _ \Rightarrow \text{False})) \wedge \text{safe_formula } \psi$)

| safe_formula (Until φ I ψ) = ($\text{fv } \varphi \subseteq \text{fv } \psi \wedge$

(safe_formula $\varphi \vee (\text{case } \varphi \text{ of Neg } \varphi' \Rightarrow \text{safe_formula } \varphi' \mid _ \Rightarrow \text{False})) \wedge \text{safe_formula } \psi$)

| safe_formula (MatchP I r) = Regex.safe_regex fv ($\lambda g \varphi. \text{safe_formula } \varphi \vee$

($g = \text{Lax} \wedge (\text{case } \varphi \text{ of Neg } \varphi' \Rightarrow \text{safe_formula } \varphi' \mid _ \Rightarrow \text{False})))$ Past Strict r

| safe_formula (MatchF I r) = Regex.safe_regex fv ($\lambda g \varphi. \text{safe_formula } \varphi \vee$

($g = \text{Lax} \wedge (\text{case } \varphi \text{ of Neg } \varphi' \Rightarrow \text{safe_formula } \varphi' \mid _ \Rightarrow \text{False})))$ Futu Strict r

abbreviation safe_regex \equiv Regex.safe_regex fv ($\lambda g \varphi. \text{safe_formula } \varphi \vee$

($g = \text{Lax} \wedge (\text{case } \varphi \text{ of Neg } \varphi' \Rightarrow \text{safe_formula } \varphi' \mid _ \Rightarrow \text{False})))$)

lemma safe_regex_safe_formula:

safe_regex m g r $\implies \varphi \in \text{Regex.atms } r \implies \text{safe_formula } \varphi \vee$

($\exists \psi. \varphi = \text{Neg } \psi \wedge \text{safe_formula } \psi$)

by (cases g) (auto dest!: safe_regex_safe[rotated] split: formula.splits[where formula= φ])

lemma safe_abbrevs[simp]: safe_formula TT safe_formula FF
 unfolding TT_def FF_def by auto

definition safe_neg :: formula \Rightarrow bool **where**
 safe_neg $\varphi \longleftrightarrow (\neg \text{safe_formula } \varphi \longrightarrow \text{safe_formula } (\text{remove_neg } \varphi))$

definition atms :: formula Regex.regex \Rightarrow formula set **where**
 atms r = ($\bigcup \varphi \in \text{Regex.atms } r$.
 if safe_formula φ then $\{\varphi\}$ else case φ of Neg $\varphi' \Rightarrow \{\varphi'\} \mid _ \Rightarrow \{\}$)

lemma atms_simps[simp]:
 atms (Regex.Skip n) = $\{\}$
 atms (Regex.Test φ) = (if safe_formula φ then $\{\varphi\}$ else case φ of Neg $\varphi' \Rightarrow \{\varphi'\} \mid _ \Rightarrow \{\}$)
 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
 unfolding atms_def by auto

lemma finite_atms[simp]: finite (atms r)
 by (induct r) (auto split: formula.splits)

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

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: $\bigwedge c d. P (\text{Eq } (\text{Const } c) (\text{Const } d))$
and Eq_Var1: $\bigwedge c x. P (\text{Eq } (\text{Const } c) (\text{Var } x))$
and Eq_Var2: $\bigwedge c x. P (\text{Eq } (\text{Var } x) (\text{Const } c))$
and neg_Var: $\bigwedge x. P (\text{Neg } (\text{Eq } (\text{Var } x) (\text{Var } x)))$
and Pred: $\bigwedge e ts. \forall t \in \text{set } ts. \text{is_Var } t \vee \text{is_Const } t \Longrightarrow P (\text{Pred } e \text{ } ts)$
and Let: $\bigwedge p \varphi \psi. \{0..<nfv \varphi\} \subseteq fv \varphi \Longrightarrow \text{safe_formula } \varphi \Longrightarrow \text{safe_formula } \psi \Longrightarrow P \varphi \Longrightarrow P \psi$
 $\Longrightarrow P (\text{Let } p \varphi \psi)$
and And_assign: $\bigwedge \varphi \psi. \text{safe_formula } \varphi \Longrightarrow \text{safe_assignment } (fv \varphi) \psi \Longrightarrow P \varphi \Longrightarrow P (\text{And } \varphi \psi)$
and And_safe: $\bigwedge \varphi \psi. \text{safe_formula } \varphi \Longrightarrow \neg \text{safe_assignment } (fv \varphi) \psi \Longrightarrow \text{safe_formula } \psi \Longrightarrow$
 $P \varphi \Longrightarrow P \psi \Longrightarrow P (\text{And } \varphi \psi)$
and And_constraint: $\bigwedge \varphi \psi. \text{safe_formula } \varphi \Longrightarrow \neg \text{safe_assignment } (fv \varphi) \psi \Longrightarrow \neg \text{safe_formula } \psi$
 \Longrightarrow
 $fv \psi \subseteq fv \varphi \Longrightarrow \text{is_constraint } \psi \Longrightarrow P \varphi \Longrightarrow P (\text{And } \varphi \psi)$
and And_Not: $\bigwedge \varphi \psi. \text{safe_formula } \varphi \Longrightarrow \neg \text{safe_assignment } (fv \varphi) (\text{Neg } \psi) \Longrightarrow \neg \text{safe_formula}$
 $(\text{Neg } \psi) \Longrightarrow$
 $fv (\text{Neg } \psi) \subseteq fv \varphi \Longrightarrow \neg \text{is_constraint } (\text{Neg } \psi) \Longrightarrow \text{safe_formula } \psi \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (\text{And}$
 $\varphi (\text{Neg } \psi))$
and Ands: $\bigwedge l \text{ pos neg. } (\text{pos}, \text{neg}) = \text{partition safe_formula } l \Longrightarrow \text{pos} \neq [] \Longrightarrow$
 $\text{list_all safe_formula pos} \Longrightarrow \text{list_all safe_formula } (\text{map remove_neg neg}) \Longrightarrow$
 $(\bigcup \varphi \in \text{set neg. } fv \varphi) \subseteq (\bigcup \varphi \in \text{set pos. } fv \varphi) \Longrightarrow$
 $\text{list_all } P \text{ pos} \Longrightarrow \text{list_all } P (\text{map remove_neg neg}) \Longrightarrow P (\text{Ands } l)$
and Neg: $\bigwedge \varphi. fv \varphi = \{\} \Longrightarrow \text{safe_formula } \varphi \Longrightarrow P \varphi \Longrightarrow P (\text{Neg } \varphi)$
and Or: $\bigwedge \varphi \psi. fv \psi = fv \varphi \Longrightarrow \text{safe_formula } \varphi \Longrightarrow \text{safe_formula } \psi \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (\text{Or}$
 $\varphi \psi)$
and Exists: $\bigwedge \varphi. \text{safe_formula } \varphi \Longrightarrow P \varphi \Longrightarrow P (\text{Exists } \varphi)$
and Agg: $\bigwedge y \omega b f \varphi. y + b \notin fv \varphi \Longrightarrow \{0..<b\} \subseteq fv \varphi \Longrightarrow fv_trm f \subseteq fv \varphi \Longrightarrow$
 $\text{safe_formula } \varphi \Longrightarrow P \varphi \Longrightarrow P (\text{Agg } y \omega b f \varphi)$
and Prev: $\bigwedge I \varphi. \text{safe_formula } \varphi \Longrightarrow P \varphi \Longrightarrow P (\text{Prev } I \varphi)$

```

and Next:  $\bigwedge I \varphi. \text{safe\_formula } \varphi \implies P \varphi \implies P (\text{Next } I \varphi)$ 
and Since:  $\bigwedge \varphi I \psi. \text{fv } \varphi \subseteq \text{fv } \psi \implies \text{safe\_formula } \varphi \implies \text{safe\_formula } \psi \implies P \varphi \implies P \psi \implies P$ 
(Since  $\varphi I \psi$ )
and Not_Since:  $\bigwedge \varphi I \psi. \text{fv } (\text{Neg } \varphi) \subseteq \text{fv } \psi \implies \text{safe\_formula } \varphi \implies$ 
 $\neg \text{safe\_formula } (\text{Neg } \varphi) \implies \text{safe\_formula } \psi \implies P \varphi \implies P \psi \implies P (\text{Since } (\text{Neg } \varphi) I \psi)$ 
and Until:  $\bigwedge \varphi I \psi. \text{fv } \varphi \subseteq \text{fv } \psi \implies \text{safe\_formula } \varphi \implies \text{safe\_formula } \psi \implies P \varphi \implies P \psi \implies P$ 
(Until  $\varphi I \psi$ )
and Not_Until:  $\bigwedge \varphi I \psi. \text{fv } (\text{Neg } \varphi) \subseteq \text{fv } \psi \implies \text{safe\_formula } \varphi \implies$ 
 $\neg \text{safe\_formula } (\text{Neg } \varphi) \implies \text{safe\_formula } \psi \implies P \varphi \implies P \psi \implies P (\text{Until } (\text{Neg } \varphi) I \psi)$ 
and MatchP:  $\bigwedge I r. \text{safe\_regex Past Strict } r \implies \forall \varphi \in \text{atms } r. P \varphi \implies P (\text{MatchP } I r)$ 
and MatchF:  $\bigwedge I r. \text{safe\_regex Futu Strict } r \implies \forall \varphi \in \text{atms } r. P \varphi \implies P (\text{MatchF } I r)$ 
shows  $P \varphi$ 
using assms(1) proof (induction  $\varphi$  rule: safe_formula.induct)
case (1 t1 t2)
then show ?case using Eq_Const Eq_Var1 Eq_Var2 by (auto simp: trm.is_Const_def trm.is_Var_def)
next
case (9  $\varphi \psi$ )
from  $\langle \text{safe\_formula } (\text{And } \varphi \psi) \rangle$  have safe_formula  $\varphi$  by simp
from  $\langle \text{safe\_formula } (\text{And } \varphi \psi) \rangle$  consider
(a) safe_assignment (fv  $\varphi$ )  $\psi$ 
| (b)  $\neg \text{safe\_assignment } (\text{fv } \varphi) \psi \text{ safe\_formula } \psi$ 
| (c)  $\text{fv } \psi \subseteq \text{fv } \varphi \neg \text{safe\_assignment } (\text{fv } \varphi) \psi \neg \text{safe\_formula } \psi \text{ is\_constraint } \psi$ 
| (d)  $\psi'$  where  $\text{fv } \psi \subseteq \text{fv } \varphi \neg \text{safe\_assignment } (\text{fv } \varphi) \psi \neg \text{safe\_formula } \psi \neg \text{is\_constraint } \psi$ 
 $\psi = \text{Neg } \psi' \text{ safe\_formula } \psi'$ 
by (cases  $\psi$ ) auto
then show ?case proof cases
case a
then show ?thesis using 9.IH  $\langle \text{safe\_formula } \varphi \rangle$  by (intro And_assign)
next
case b
then show ?thesis using 9.IH  $\langle \text{safe\_formula } \varphi \rangle$  by (intro And_safe)
next
case c
then show ?thesis using 9.IH  $\langle \text{safe\_formula } \varphi \rangle$  by (intro And_constraint)
next
case d
then show ?thesis using 9.IH  $\langle \text{safe\_formula } \varphi \rangle$  by (blast intro!: And_Not)
qed
next
case (10 l)
obtain pos neg where posneg: (pos, neg) = partition safe_formula l by simp
have  $\text{pos} \neq []$  using 10.prems posneg by simp
moreover have list_all safe_formula pos using posneg by (simp add: list.pred_set)
moreover have safe_remove_neg: list_all safe_formula (map remove_neg neg) using 10.prems posneg
by auto
moreover have list_all P pos
using posneg 10.IH(1) by (simp add: list_all_iff)
moreover have list_all P (map remove_neg neg)
using 10.IH(2)[OF posneg] safe_remove_neg by (simp add: list_all_iff)
ultimately show ?case using 10.IH(1) 10.prems Ands posneg by simp
next
case (15  $\varphi I \psi$ )
then show ?case
proof (cases  $\varphi$ )
case (Ands l)
then show ?thesis using 15.IH(1) 15.IH(3) 15.prems Since by auto
qed (auto 0 3 elim!: disjE_Not2 intro: Since Not_Since)
next

```

```

case (16  $\varphi$  I  $\psi$ )
then show ?case
proof (cases  $\varphi$ )
  case (Ands l)
    then show ?thesis using 16.IH(1) 16.IH(3) 16.premys Until by auto
qed (auto 0 3 elim!: disjE_Not2 intro: Until Not_Until)
next
case (17 I r)
then show ?case
  by (intro MatchP) (auto simp: atms_def dest: safe_regex_safe_formula split: if_splits)
next
case (18 I r)
then show ?case
  by (intro MatchF) (auto simp: atms_def dest: safe_regex_safe_formula split: if_splits)
qed (auto simp: assms)

```

```

lemma safe_formula_NegD:
  safe_formula (Formula.Neg  $\varphi$ )  $\implies$  fv  $\varphi$  = {}  $\vee$  ( $\exists x. \varphi$  = Formula.Eq (Formula.Var x) (Formula.Var x))
by (induct Formula.Neg  $\varphi$  rule: safe_formula_induct) auto

```

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)
| matches v (Let p  $\varphi$   $\psi$ ) e =
  (( $\exists v'. \text{matches } v' \varphi e \wedge \text{matches } v \psi (p, v')$ )  $\vee$ 
   fst e  $\neq$  p  $\wedge$  matches v  $\psi e$ )
| matches v (Eq _ _) e = False
| matches v (Less _ _) e = False
| matches v (LessEq _ _) e = False
| matches v (Neg  $\varphi$ ) e = matches v  $\varphi e$ 
| matches v (Or  $\varphi$   $\psi$ ) e = (matches v  $\varphi e \vee$  matches v  $\psi e$ )
| matches v (And  $\varphi$   $\psi$ ) e = (matches v  $\varphi e \wedge$  matches v  $\psi e$ )
| matches v (Ands l) e = ( $\exists \varphi \in \text{set } l. \text{matches } v \varphi e$ )
| matches v (Exists  $\varphi$ ) e = ( $\exists z. \text{matches } (z \# v) \varphi e$ )
| matches v (Agg y  $\omega$  b f  $\varphi$ ) e = ( $\exists zs. \text{length } zs = b \wedge \text{matches } (zs @ v) \varphi e$ )
| matches v (Prev I  $\varphi$ ) e = matches v  $\varphi e$ 
| matches v (Next I  $\varphi$ ) e = matches v  $\varphi e$ 
| matches v (Since  $\varphi$  I  $\psi$ ) e = (matches v  $\varphi e \vee$  matches v  $\psi e$ )
| matches v (Until  $\varphi$  I  $\psi$ ) e = (matches v  $\varphi e \vee$  matches v  $\psi e$ )
| matches v (MatchP I r) e = ( $\exists \varphi \in \text{Regex.atms } r. \text{matches } v \varphi e$ )
| matches v (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 (induct  $\varphi$  arbitrary: v v' e)
  case (Pred n ts)
  show ?case
  by (simp cong: map_cong eval_trm_fv_cong[OF Pred(1)][simplified, THEN bspec])
next
case (Let p b  $\varphi$   $\psi$ )
then show ?case
  by (cases e) (auto 11 0)
next
case (Ands l)
have  $\bigwedge \varphi. \varphi \in (\text{set } l) \implies \text{matches } v \varphi e = \text{matches } v' \varphi e$ 

```

```

proof –
  fix  $\varphi$  assume  $\varphi \in (\text{set } l)$ 
  then have  $\text{fv } \varphi \subseteq \text{fv } (And\ s\ l)$  using  $\text{fv\_subset\_And\ s}$  by  $\text{blast}$ 
  then have  $\forall x \in \text{fv } \varphi. v!x = v'!x$  using  $\text{And\ s.prem\ s}$  by  $\text{blast}$ 
  then show  $\text{matches } v\ \varphi\ e = \text{matches } v'\ \varphi\ e$  using  $\text{And\ s.hyps } \langle \varphi \in \text{set } l \rangle$  by  $\text{blast}$ 
qed
then show  $?case$  by  $\text{simp}$ 
next
case  $(\text{Exists } \varphi)$ 
then show  $?case$  unfolding  $\text{matches.simp\ s}$  by  $(\text{intro } \text{iff\_exI}) (\text{simp } \text{add: } \text{fvi\_Suc } \text{nth\_Cons'})$ 
next
case  $(\text{Agg } y\ \omega\ b\ f\ \varphi)$ 
have  $\text{matches } (zs @ v)\ \varphi\ e = \text{matches } (zs @ v')\ \varphi\ e$  if  $\text{length } zs = b$  for  $zs$ 
  using that  $\text{Agg.prem\ s}$  by  $(\text{simp } \text{add: } \text{Agg.hyps}[\text{where } v=zs @ v \text{ and } v'=zs @ v]$ 
     $\text{nth\_append } \text{fvi\_iff\_fv}(1)[\text{where } b=b])$ 
  then show  $?case$  by  $\text{auto}$ 
qed  $(\text{auto } 9\ 0 \text{ simp } \text{add: } \text{nth\_Cons'}\ \text{fv\_regex\_alt})$ 

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

lemma  $\text{sat\_slice\_strong}$ :
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  $\text{relevant\_events } \varphi\ S - \{e. \text{fst } e \in \text{dom } V\} \subseteq E \implies$ 
   $\text{sat } \sigma\ V\ v\ i\ \varphi \longleftrightarrow \text{sat } (\text{map\_}\Gamma\ (\lambda D. D \cap E)\ \sigma)\ V'\ v\ i\ \varphi$ 
using  $\text{assms}$ 
proof  $(\text{induction } \varphi \text{ arbitrary: } V\ V'\ v\ S\ i)$ 
case  $(\text{Pred } r\ ts)$ 
show  $?case$  proof  $(\text{cases } V\ r)$ 
  case  $\text{None}$ 
  then have  $V'\ r = \text{None}$  using  $\langle \text{dom } V = \text{dom } V' \rangle$  by  $\text{auto}$ 
  with  $\text{None } \text{Pred}(1,2)$  show  $?thesis$  by  $(\text{auto } \text{simp: } \text{domIff } \text{dest!}: \text{subsetD})$ 
next
case  $(\text{Some } a)$ 
moreover obtain  $a'$  where  $V'\ r = \text{Some } a'$  using  $\text{Some } \langle \text{dom } V = \text{dom } V' \rangle$  by  $\text{auto}$ 
moreover have  $(\text{map } (\text{eval\_trm } v)\ ts \in \text{the } (V\ r)\ i) = (\text{map } (\text{eval\_trm } v)\ ts \in \text{the } (V'\ r)\ i)$ 
  using  $\text{Some } \text{Pred}(2,4)$  by  $(\text{fastforce } \text{intro: } \text{domI})$ 
ultimately show  $?thesis$  by  $\text{simp}$ 
qed
next
case  $(\text{Let } p\ \varphi\ \psi)$ 
from  $\text{Let.prem\ s}$  show  $?case$  unfolding  $\text{sat.simp\ s}$ 
proof  $(\text{intro } \text{Let}(2)[\text{of } S], \text{goal\_cases } \text{relevant } v\ \text{dom } V)$ 
case  $(V\ p'\ v'\ i)$ 
then show  $?case$ 
proof  $(\text{cases } p' = p)$ 
case  $[\text{simp}]: \text{True}$ 
with  $V$  show  $?thesis$ 
  unfolding  $\text{fun\_upd\_apply } \text{eqTrueI}[OF\ \text{True}] \text{ if\_True } \text{option.sel } \text{mem\_Collect\_eq}$ 
proof  $(\text{intro } \text{ex\_cong } \text{conj\_cong } \text{refl } \text{Let}(1)[\text{where } S=\{v'. (\exists v \in S. \text{matches } v\ \psi\ (p, v'))\} \text{ and } V=V],$ 
   $\text{goal\_cases } \text{relevant}'\ v'\ \text{dom}'\ V')$ 
case  $\text{relevant}'$ 
then show  $?case$ 
  by  $(\text{elim } \text{subset\_trans}[\text{rotated}]) (\text{auto } \text{simp: } \text{set\_eq\_iff})$ 
next
case  $(V'\ p'\ v'\ i)$ 
then show  $?case$ 

```

```

      by (intro V(4)) (auto simp: set_eq_iff)
    qed auto
  next
    case False
    with V(2,3,5,6) show ?thesis
      unfolding fun_upd_apply eq_False[THEN iffD2, OF False] if_False
      by (intro V(4)) (auto simp: False)
    qed
  qed (auto simp: dom_def)
next
  case (Or  $\varphi \psi$ )
  show ?case using Or.IH[of S V v V'] Or.premss
    by (auto simp: Collect_disj_eq Int_Un_distrib subset_iff)
next
  case (And  $\varphi \psi$ )
  show ?case using And.IH[of S V v V'] And.premss
    by (auto simp: Collect_disj_eq Int_Un_distrib subset_iff)
next
  case (Ands l)
  obtain relevant_events (Ands l) S - {e. fst e ∈ dom V} ⊆ E v ∈ S using Ands.premss(1) Ands.premss(2)
  by blast
  then have {e. S ∩ {v. matches v (Ands l) e} ≠ {}} - {e. fst e ∈ dom V} ⊆ E by simp
  have  $\bigwedge \varphi. \varphi \in \text{set } l \implies \text{sat } \sigma V v i \varphi \longleftrightarrow \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) V' v i \varphi$ 
  proof -
    fix  $\varphi$  assume  $\varphi \in \text{set } l$ 
    have relevant_events  $\varphi$  S = {e. S ∩ {v. matches v  $\varphi$  e} ≠ {}} by simp
    have {e. S ∩ {v. matches v  $\varphi$  e} ≠ {}} ⊆ {e. S ∩ {v. matches v (Ands l) e} ≠ {}} (is ?A ⊆ ?B)
    proof (rule subsetI)
      fix e assume e ∈ ?A
      then have S ∩ {v. matches v  $\varphi$  e} ≠ {} by blast
      moreover have S ∩ {v. matches v (Ands l) e} ≠ {}
      proof -
        obtain v where v ∈ S matches v  $\varphi$  e using calculation by blast
        then show ?thesis using  $\langle \varphi \in \text{set } l \rangle$  by auto
      qed
    qed
    then show e ∈ ?B by blast
  qed
  then have relevant_events  $\varphi$  S - {e. fst e ∈ dom V} ⊆ E using Ands.premss(1) by auto
  then show  $\text{sat } \sigma V v i \varphi \longleftrightarrow \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) V' v i \varphi$ 
    using Ands.premss(2,3)  $\langle \varphi \in \text{set } l \rangle$ 
    by (intro Ands.IH[of  $\varphi$  S V v V' i] Ands.premss(4)) auto
  qed
  show ?case using  $\langle \bigwedge \varphi. \varphi \in \text{set } l \implies \text{sat } \sigma V v i \varphi = \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) V' v i \varphi \rangle$  sat_Ands
  by blast
next
  case (Exists  $\varphi$ )
  have  $\text{sat } \sigma V (z \# v) i \varphi = \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) V' (z \# v) i \varphi$  for z
    using Exists.premss(1-3) by (intro Exists.IH[where S={z # v | v. v ∈ S}] Exists.premss(4)) auto
  then show ?case by simp
next
  case (Agg y  $\omega$  b f  $\varphi$ )
  have  $\text{sat } \sigma V (zs @ v) i \varphi = \text{sat } (\text{map\_}\Gamma (\lambda D. D \cap E) \sigma) V' (zs @ v) i \varphi$  if length zs = b for zs
    using that Agg.premss(1-3) by (intro Agg.IH[where S={zs @ v | v. v ∈ S}] Agg.premss(4)) auto
  then show ?case by (simp cong: conj_cong)
next
  case (Prev I  $\varphi$ )
  then show ?case by (auto cong: nat.case_cong)
next

```

```

    case (Next I  $\varphi$ )
  then show ?case by simp
next
  case (Since  $\varphi$  I  $\psi$ )
  show ?case using Since.IH[of S V] Since.prem
  by (auto simp: Collect_disj_eq Int_Un_distrib subset_iff)
next
  case (Until  $\varphi$  I  $\psi$ )
  show ?case using Until.IH[of S V] Until.prem
  by (auto simp: Collect_disj_eq Int_Un_distrib subset_iff)
next
  case (MatchP I r)
  from MatchP.prem(1-3) have Regex.match (sat  $\sigma$  V v) r = Regex.match (sat (map_Γ (λD. D ∩ E)
 $\sigma$ ) V' v) r
  by (intro Regex.match_fv_cong MatchP(1)[of _ S V v] MatchP.prem(4)) auto
  then show ?case
  by auto
next
  case (MatchF I r)
  from MatchF.prem(1-3) have Regex.match (sat  $\sigma$  V v) r = Regex.match (sat (map_Γ (λD. D ∩ E)
 $\sigma$ ) V' v) r
  by (intro Regex.match_fv_cong MatchF(1)[of _ S V v] MatchF.prem(4)) auto
  then show ?case
  by auto
qed simp_all

```

4.5 Translation to n-ary conjunction

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

```

  get_and_list (Ands l) = l
| get_and_list  $\varphi$  = [ $\varphi$ ]

```

lemma *fv_get_and*: $(\bigcup x \in (\text{set } (\text{get_and_list } \varphi)). \text{fv } b \ x) = \text{fv } b \ \varphi$
by (induction φ rule: get_and_list.induct) simp_all

lemma *safe_get_and*: *safe_formula* $\varphi \implies \text{list_all } \text{safe_neg } (\text{get_and_list } \varphi)$
by (induction φ rule: get_and_list.induct) (simp_all add: safe_neg_def list_all_iff)

lemma *sat_get_and*: $\text{sat } \sigma \ V \ v \ i \ \varphi \iff \text{list_all } (\text{sat } \sigma \ V \ v \ i) (\text{get_and_list } \varphi)$
by (induction φ rule: get_and_list.induct) (simp_all add: list_all_iff)

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

```

  convert_multiway (Neg  $\varphi$ ) = Neg (convert_multiway  $\varphi$ )
| convert_multiway (Or  $\varphi$   $\psi$ ) = Or (convert_multiway  $\varphi$ ) (convert_multiway  $\psi$ )
| convert_multiway (And  $\varphi$   $\psi$ ) = (if safe_assignment (fv  $\varphi$ )  $\psi$  then
  And (convert_multiway  $\varphi$ )  $\psi$ 
  else if safe_formula  $\psi$  then
  Ands (get_and_list (convert_multiway  $\varphi$ ) @ get_and_list (convert_multiway  $\psi$ ))
  else if is_constraint  $\psi$  then
  And (convert_multiway  $\varphi$ )  $\psi$ 
  else Ands (convert_multiway  $\psi$  # get_and_list (convert_multiway  $\varphi$ )))
| convert_multiway (Exists  $\varphi$ ) = Exists (convert_multiway  $\varphi$ )
| convert_multiway (Agg y  $\omega$  b f  $\varphi$ ) = Agg y  $\omega$  b f (convert_multiway  $\varphi$ )
| convert_multiway (Prev I  $\varphi$ ) = Prev I (convert_multiway  $\varphi$ )
| convert_multiway (Next I  $\varphi$ ) = Next I (convert_multiway  $\varphi$ )
| convert_multiway (Since  $\varphi$  I  $\psi$ ) = Since (convert_multiway  $\varphi$ ) I (convert_multiway  $\psi$ )
| convert_multiway (Until  $\varphi$  I  $\psi$ ) = Until (convert_multiway  $\varphi$ ) I (convert_multiway  $\psi$ )
| convert_multiway (MatchP I r) = MatchP I (Regex.map_regex convert_multiway r)

```

| *convert_multiway* (MatchF I r) = MatchF I (Regex.map_regex *convert_multiway* r)
| *convert_multiway* φ = φ

abbreviation *convert_multiway_regex* \equiv Regex.map_regex *convert_multiway*

lemma *fv_safe_get_and*:

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

proof (*induction* φ *rule*: *get_and_list.induct*)

case (1 l)

obtain *pos neg* **where** *posneg*: (*pos*, *neg*) = *partition safe_formula l by simp*

have *get_and_list* (*And*s l) = l **by** *simp*

have *sub*: ($\bigcup x \in \text{set } \text{neg. fv } x \subseteq \bigcup x \in \text{set } \text{pos. fv } x$) **using** 1.prem*s* *posneg* **by** *simp*

then have *fv* (*And*s l) $\subseteq (\bigcup x \in \text{set } \text{pos. fv } x)$

proof –

have *fv* (*And*s l) = ($\bigcup x \in \text{set } \text{pos. fv } x$) \cup ($\bigcup x \in \text{set } \text{neg. fv } x$) **using** *posneg* **by** *auto*

then show *?thesis* **using** *sub* **by** *simp*

qed

then show *?case* **using** *posneg* **by** *auto*

qed *auto*

lemma *ex_safe_get_and*:

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

proof (*induction* φ *rule*: *get_and_list.induct*)

case (1 l)

have *get_and_list* (*And*s l) = l **by** *simp*

obtain *pos neg* **where** *posneg*: (*pos*, *neg*) = *partition safe_formula l by simp*

then have *pos* $\neq []$ **using** 1.prem*s* **by** *simp*

then obtain *x* **where** $x \in \text{set } \text{pos}$ **by** *fastforce*

then show *?case* **using** *posneg* **using** *Bex_set_list_ex* **by** *fastforce*

qed *simp_all*

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

by (*cases* φ) *simp_all*

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

by (*cases* φ) (*auto elim*: *case_NegE*)

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

proof (*induction* φ *arbitrary*: *b* *rule*: *safe_formula.induct*)

case (9 $\varphi \psi$)

then show *?case* **by** (*cases* ψ) (*auto simp*: *fv_get_and Un_commute*)

next

case (15 $\varphi I \psi$)

show *?case* **proof** (*cases* *safe_formula* φ)

case *True*

with 15 **show** *?thesis* **by** *simp*

next

case *False*

with 15.prem*s* **obtain** φ' **where** $\varphi = \text{Neg } \varphi'$ **by** (*simp split*: *formula.splits*)

with *False* 15 **show** *?thesis* **by** *simp*

qed

next

case (16 $\varphi I \psi$)

show *?case* **proof** (*cases* *safe_formula* φ)

case *True*

with 16 **show** *?thesis* **by** *simp*

```

next
  case False
  with 16.premis obtain  $\varphi'$  where  $\varphi = \text{Neg } \varphi'$  by (simp split: formula.splits)
  with False 16 show ?thesis by simp
qed
next
case (17 I r)
then show ?case
  unfolding convert_multiway.simps fvi.simps fv_regex_alt regex.set_map image_image
  by (intro arg_cong[where f=Union, OF image_cong[OF refl]])
    (auto dest!: safe_regex_safe_formula)
next
case (18 I r)
then show ?case
  unfolding convert_multiway.simps fvi.simps fv_regex_alt regex.set_map image_image
  by (intro arg_cong[where f=Union, OF image_cong[OF refl]])
    (auto dest!: safe_regex_safe_formula)
qed (auto simp del: convert_multiway.simps(3))

lemma get_and_nonempty:
  assumes safe_formula  $\varphi$ 
  shows get_and_list  $\varphi \neq []$ 
  using assms by (induction  $\varphi$ ) auto

lemma future_bounded_get_and:
  list_all future_bounded (get_and_list  $\varphi$ ) = future_bounded  $\varphi$ 
  by (induction  $\varphi$ ) simp_all

lemma safe_convert_multiway: safe_formula  $\varphi \implies$  safe_formula (convert_multiway  $\varphi$ )
proof (induction  $\varphi$  rule: safe_formula_induct)
  case (And_safe  $\varphi \psi$ )
  let ?a = And  $\varphi \psi$ 
  let ?b = convert_multiway ?a
  let ?c $\varphi$  = convert_multiway  $\varphi$ 
  let ?c $\psi$  = convert_multiway  $\psi$ 
  have b_def: ?b = Ands (get_and_list ?c $\varphi$  @ get_and_list ?c $\psi$ )
    using And_safe by simp
  show ?case proof -
    let ?l = get_and_list ?c $\varphi$  @ get_and_list ?c $\psi$ 
    obtain pos neg where posneg: (pos, neg) = partition safe_formula ?l by simp
    then have list_all_safe_formula pos by (auto simp: list_all_iff)
    have lsafe_neg: list_all safe_neg ?l
      using And_safe ⟨safe_formula  $\varphi$ ⟩ ⟨safe_formula  $\psi$ ⟩
      by (simp add: safe_get_and)
    then have list_all_safe_formula (map remove_neg neg)
  proof -
    have  $\bigwedge x. x \in \text{set } \text{neg} \implies \text{safe\_formula } (\text{remove\_neg } x)$ 
  proof -
    fix  $x$  assume  $x \in \text{set } \text{neg}$ 
    then have  $\neg \text{safe\_formula } x$  using posneg by auto
    moreover have safe_neg  $x$  using lsafe_neg ⟨ $x \in \text{set } \text{neg}$ ⟩
      unfolding safe_neg_def list_all_iff partition_set[OF posneg[symmetric], symmetric]
      by simp
    ultimately show safe_formula (remove_neg  $x$ ) using safe_neg_def by blast
  qed
  then show ?thesis by (auto simp: list_all_iff)
qed

```

```

have pos_filter: pos = filter safe_formula (get_and_list ?cφ @ get_and_list ?cψ)
  using posneg by simp
have (⋃ x∈set neg. fv x) ⊆ (⋃ x∈set pos. fv x)
proof -
  have 1: fv ?cφ ⊆ (⋃ x∈(set (filter safe_formula (get_and_list ?cφ))). fv x) (is _ ⊆ ?fvφ)
    using And_safe ⟨safe_formula φ⟩ by (blast intro!: fv_safe_get_and)
  have 2: fv ?cψ ⊆ (⋃ x∈(set (filter safe_formula (get_and_list ?cψ))). fv x) (is _ ⊆ ?fvψ)
    using And_safe ⟨safe_formula ψ⟩ by (blast intro!: fv_safe_get_and)
  have (⋃ x∈set neg. fv x) ⊆ fv ?cφ ∪ fv ?cψ proof -
    have ⋃ (fv ' set neg) ⊆ ⋃ (fv ' (set pos ∪ set neg))
      by simp
    also have ... ⊆ fv (convert_multiway φ) ∪ fv (convert_multiway ψ)
      unfolding partition_set[OF posneg[symmetric], simplified]
      by (simp add: fv_get_and)
    finally show ?thesis .
qed
then have (⋃ x∈set neg. fv x) ⊆ ?fvφ ∪ ?fvψ using 1 2 by blast
then show ?thesis unfolding pos_filter by simp
qed
have pos ≠ []
proof -
  obtain x where x ∈ set (get_and_list ?cφ) safe_formula x
    using And_safe ⟨safe_formula φ⟩ ex_safe_get_and by (auto simp: list_ex_iff)
  then show ?thesis
    unfolding pos_filter by (auto simp: filter_empty_conv)
qed
then show ?thesis unfolding b_def
  using ⟨⋃ (fv ' set neg) ⊆ ⋃ (fv ' set pos)⟩ ⟨list_all safe_formula (map remove_neg neg)⟩
  ⟨list_all safe_formula pos⟩ posneg
  by simp
qed
next
case (And_Not φ ψ)
let ?a = And φ (Neg ψ)
let ?b = convert_multiway ?a
let ?cφ = convert_multiway φ
let ?cψ = convert_multiway ψ
have b_def: ?b = Ands (Neg ?cψ # get_and_list ?cφ)
  using And_Not by simp
show ?case proof -
  let ?l = Neg ?cψ # get_and_list ?cφ
  note ⟨safe_formula ?cφ⟩
  then have list_all safe_neg (get_and_list ?cφ) by (simp add: safe_get_and)
  moreover have safe_neg (Neg ?cψ)
    using ⟨safe_formula ?cψ⟩ by (simp add: safe_neg_def)
  then have lsafe_neg: list_all safe_neg ?l using calculation by simp
  obtain pos neg where posneg: (pos, neg) = partition safe_formula ?l by simp
  then have list_all safe_formula pos by (auto simp: list_all_iff)
  then have list_all safe_formula (map remove_neg neg)
proof -
  have ∧x. x ∈ (set neg) ⇒ safe_formula (remove_neg x)
proof -
  fix x assume x ∈ set neg
  then have ¬ safe_formula x using posneg by (auto simp del: filter_simps)
  moreover have safe_neg x using lsafe_neg ⟨x ∈ set neg⟩
    unfolding safe_neg_def list_all_iff partition_set[OF posneg[symmetric], symmetric]
    by simp
  ultimately show safe_formula (remove_neg x) using safe_neg_def by blast

```

```

qed
then show ?thesis using Ball_set_list_all by force
qed

have pos_filter: pos = filter safe_formula ?l
  using posneg by simp
have neg_filter: neg = filter (Not ∘ safe_formula) ?l
  using posneg by simp
have (⋃ x ∈ (set neg). fv x) ⊆ (⋃ x ∈ (set pos). fv x)
proof -
  have fv_neg: (⋃ x ∈ (set neg). fv x) ⊆ (⋃ x ∈ (set ?l). fv x) using posneg by auto
  have (⋃ x ∈ (set ?l). fv x) ⊆ fv ?cφ ∪ fv ?cψ
    using ⟨safe_formula φ⟩ ⟨safe_formula ψ⟩
    by (simp add: fv_get_and fv_convert_multiway)
  also have fv ?cφ ∪ fv ?cψ ⊆ fv ?cφ
    using ⟨safe_formula φ⟩ ⟨safe_formula ψ⟩ ⟨fv (Neg ψ) ⊆ fv φ⟩
    by (simp add: fv_convert_multiway[symmetric])
  finally have (⋃ x ∈ (set neg). fv x) ⊆ fv ?cφ
    using fv_neg unfolding neg_filter by blast
  then show ?thesis
    unfolding pos_filter
    using fv_safe_get_and[OF And_Not.IH(1)]
    by auto
qed
have pos ≠ []
proof -
  obtain x where x ∈ set (get_and_list ?cφ) safe_formula x
    using And_Not.IH ⟨safe_formula φ⟩ ex_safe_get_and by (auto simp: list_ex_iff)
  then show ?thesis
    unfolding pos_filter by (auto simp: filter_empty_conv)
qed
then show ?thesis unfolding b_def
  using ⟨⋃ (fv ' set neg) ⊆ ⋃ (fv ' set pos)⟩ ⟨list_all safe_formula (map remove_neg neg)⟩
  ⟨list_all safe_formula pos⟩ posneg
  by simp
qed
next
case (Neg φ)
have safe_formula (Neg φ') ↔ safe_formula φ' if fv φ' = {} for φ'
  using that by (cases Neg φ' rule: safe_formula.cases) simp_all
with Neg show ?case by (simp add: fv_convert_multiway)
next
case (MatchP I r)
then show ?case
  by (auto 0 3 simp: atms_def fv_convert_multiway intro!: safe_regex_map_regex
    elim!: disjE_Not2 case_NegE
    dest: safe_regex_safe_formula split: if_splits)
next
case (MatchF I r)
then show ?case
  by (auto 0 3 simp: atms_def fv_convert_multiway intro!: safe_regex_map_regex
    elim!: disjE_Not2 case_NegE
    dest: safe_regex_safe_formula split: if_splits)
qed (auto simp: fv_convert_multiway)

lemma future_bounded_convert_multiway: safe_formula φ ⇒ future_bounded (convert_multiway φ)
= future_bounded φ
proof (induction φ rule: safe_formula_induct)

```

```

case (And_safe  $\varphi$   $\psi$ )
let ?a = And  $\varphi$   $\psi$ 
let ?b = convert_multiway ?a
let ?c $\varphi$  = convert_multiway  $\varphi$ 
let ?c $\psi$  = convert_multiway  $\psi$ 
have b_def: ?b = And (get_and_list ?c $\varphi$  @ get_and_list ?c $\psi$ )
  using And_safe by simp
have future_bounded ?a = (future_bounded ?c $\varphi$   $\wedge$  future_bounded ?c $\psi$ )
  using And_safe by simp
moreover have future_bounded ?c $\varphi$  = list_all future_bounded (get_and_list ?c $\varphi$ )
  using  $\langle$ safe_formula  $\varphi$  $\rangle$  by (simp add: future_bounded_get_and safe_convert_multiway)
moreover have future_bounded ?c $\psi$  = list_all future_bounded (get_and_list ?c $\psi$ )
  using  $\langle$ safe_formula  $\psi$  $\rangle$  by (simp add: future_bounded_get_and safe_convert_multiway)
moreover have future_bounded ?b = list_all future_bounded (get_and_list ?c $\varphi$  @ get_and_list ?c $\psi$ )
  unfolding b_def by simp
ultimately show ?case by simp
next
case (And_Not  $\varphi$   $\psi$ )
let ?a = And  $\varphi$  (Neg  $\psi$ )
let ?b = convert_multiway ?a
let ?c $\varphi$  = convert_multiway  $\varphi$ 
let ?c $\psi$  = convert_multiway  $\psi$ 
have b_def: ?b = And (Neg ?c $\psi$  # get_and_list ?c $\varphi$ )
  using And_Not by simp
have future_bounded ?a = (future_bounded ?c $\varphi$   $\wedge$  future_bounded ?c $\psi$ )
  using And_Not by simp
moreover have future_bounded ?c $\varphi$  = list_all future_bounded (get_and_list ?c $\varphi$ )
  using  $\langle$ safe_formula  $\varphi$  $\rangle$  by (simp add: future_bounded_get_and safe_convert_multiway)
moreover have future_bounded ?b = list_all future_bounded (Neg ?c $\psi$  # get_and_list ?c $\varphi$ )
  unfolding b_def by (simp add: list.pred_map o_def)
ultimately show ?case by auto
next
case (MatchP I r)
then show ?case
  by (fastforce simp: atms_def regex.pred_set regex.set_map ball_Un
    elim: safe_regex_safe_formula[THEN disjE_Not2])
next
case (MatchF I r)
then show ?case
  by (fastforce simp: atms_def regex.pred_set regex.set_map ball_Un
    elim: safe_regex_safe_formula[THEN disjE_Not2])
qed auto

lemma sat_convert_multiway: safe_formula  $\varphi$   $\implies$  sat  $\sigma$  V v i (convert_multiway  $\varphi$ )  $\longleftrightarrow$  sat  $\sigma$  V v i
 $\varphi$ 
proof (induction  $\varphi$  arbitrary: v i rule: safe_formula_induct)
case (And_safe  $\varphi$   $\psi$ )
let ?a = And  $\varphi$   $\psi$ 
let ?b = convert_multiway ?a
let ?la = get_and_list (convert_multiway  $\varphi$ )
let ?lb = get_and_list (convert_multiway  $\psi$ )
let ?sat = sat  $\sigma$  V v i
have b_def: ?b = And (?la @ ?lb) using And_safe by simp
have list_all ?sat ?la  $\longleftrightarrow$  ?sat  $\varphi$  using And_safe sat_get_and by blast
moreover have list_all ?sat ?lb  $\longleftrightarrow$  ?sat  $\psi$  using And_safe sat_get_and by blast
ultimately show ?case using And_safe by (auto simp: list.pred_set)
next
case (And_Not  $\varphi$   $\psi$ )

```

```

let ?a = And  $\varphi$  (Neg  $\psi$ )
let ?b = convert_multiway ?a
let ?la = get_and_list (convert_multiway  $\varphi$ )
let ?lb = convert_multiway  $\psi$ 
let ?sat = sat  $\sigma$  V v i
have b_def: ?b = Ands (Neg ?lb # ?la) using And_Not by simp
have list_all ?sat ?la  $\longleftrightarrow$  ?sat  $\varphi$  using And_Not sat_get_and by blast
then show ?case using And_Not by (auto simp: list.pred_set)
next
case (Agg y  $\omega$  b f  $\varphi$ )
then show ?case
  by (simp add: nfv_def fv_convert_multiway cong: conj_cong)
next
case (MatchP I r)
then have Regex.match (sat  $\sigma$  V v) (convert_multiway_regex r) = Regex.match (sat  $\sigma$  V v) r
  unfolding match_map_regex
  by (intro Regex.match_fv_cong)
  (auto 0 4 simp: atms_def elim!: disjE_Not2 dest!: safe_regex_safe_formula)
then show ?case
  by auto
next
case (MatchF I r)
then have Regex.match (sat  $\sigma$  V v) (convert_multiway_regex r) = Regex.match (sat  $\sigma$  V v) r
  unfolding match_map_regex
  by (intro Regex.match_fv_cong)
  (auto 0 4 simp: atms_def elim!: disjE_Not2 dest!: safe_regex_safe_formula)
then show ?case
  by auto
qed (auto cong: nat.case_cong)

end

```

interpretation *Formula_slicer*: abstract_slicer relevant_events φ for φ .

lemma *sat_slice_iff*:
assumes $v \in S$
shows *Formula*.sat σ V v i $\varphi \longleftrightarrow$ *Formula*.sat (*Formula_slicer*.slice φ S σ) V v i φ
by (rule *sat_slice_strong*[OF *assms*]) auto

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 $\psi \Rightarrow f \psi \mid _ \Rightarrow g \varphi$) =
 $(\neg ((\exists \psi. \varphi = \text{formula.Neg } \psi \wedge \neg P (f \psi)) \vee ((\neg \text{Formula.is_Neg } \varphi) \wedge \neg P (g \varphi))))$
by (cases φ ; auto simp: *Formula.is_Neg_def*)+

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 = 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*: ($\bigwedge i. i \in \text{set } bs \implies \neg i$) $\implies \text{length } bs = \text{length } as \implies$
proj_tuple bs as = *replicate* (length bs) None
by (*induction* bs as *rule*: *proj_tuple.induct*) *fastforce*+

lemma *proj_tuple_join_mask_empty*: $\text{length } as = n \implies$
proj_tuple (*join_mask* n {}) as = *replicate* n None
using *proj_tuple_replicate*[of *join_mask* n {}] **by** (*auto simp add*: *join_mask_def*)

lemma *proj_tuple_alt*: *proj_tuple* bs as = *map2* ($\lambda b a. \text{if } b \text{ then } a \text{ else None}$) bs as
by (*induction* bs as *rule*: *proj_tuple.induct*) *auto*

lemma *map2_map*: *map2* f (*map* g [0.. $\text{length } as$]) as = *map* ($\lambda i. f (g i) (as ! i)$) [0.. $\text{length } as$]
by (*rule* *nth_equality1*) *auto*

lemma *proj_tuple_join_mask_restrict*: $\text{length } as = n \implies$
proj_tuple (*join_mask* n X) as = *restrict* X as
by (*auto simp add*: *restrict_def proj_tuple_alt join_mask_def map2_map*)

lemma *wf_tuple_proj_idle*:
assumes *wf*: *wf_tuple* n X as
shows *proj_tuple* (*join_mask* n X) as = as
using *proj_tuple_join_mask_restrict*[of as n X, *unfolded restrict_idle[OF wf]*] *wf*
by (*auto simp add*: *wf_tuple_def*)

lemma *wf_tuple_change_base*:
assumes *wf*: *wf_tuple* n X as
and *mask*: *join_mask* n X = *join_mask* n Y
shows *wf_tuple* n Y as
using *wf_mask* **by** (*auto simp add*: *wf_tuple_def join_mask_def*)

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

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
using *wf_tuple_proj_idle*[OF *wf*] **by** (*auto simp add*: *proj_tuple_in_join_def*)

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}
using *assms* *proj_tuple_join_mask_restrict*[of _ n L] *join_restrict*[of t2 n R t1 L pos]
wf_tuple_length restrict_idle
by (*auto simp add*: *table_def proj_tuple_in_join_def sup.absorb1*) *fastforce*+

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}
using *assms* *proj_tuple_join_mask_restrict*[of _ n R] *join_restrict*[of t2 n R t1 L True]
wf_tuple_length restrict_idle
by (*auto simp add*: *table_def proj_tuple_in_join_def sup.absorb1 Un.absorb1*) *fastforce*+

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 $t2 - t1$)
using *join_sub*[*OF__tab*, of *pos*] *tab*(2) *proj_tuple_in_join_mask_idle*[of *n R__pos t1*]
by (*auto simp add: table_def*)

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 {})
using *join_sub*[*OF__tab*, of *pos*] *tab*(2)
by (*auto simp add: table_def proj_tuple_in_join_def wf_tuple_length proj_tuple_join_mask_empty*)

lemma *join_empty_left*: *join {} pos t* = {}
by (*auto simp add: join_def*)

lemma *join_empty_right*: *join t pos {}* = (if *pos* then {} else *t*)
by (*auto simp add: join_def*)

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'*
using *assms join_empty_left*[of *pos t'*] *join_empty_right*[of *t pos*]
join_no_cols[*OF__assms*(1), of *t' pos*] *join_eq*[of *n A t' t pos*] *join_sub*[*OF__assms*(2,1)]
join_sub'[*OF__assms*(2,1)]
by *auto+*

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* ($\lambda A X. \text{table } n A X \wedge \text{wf_set } n A$) (*A_pos* @ *A_neg*) *L*
shows $z \in \text{mmulti_join}' A_pos A_neg L \longleftrightarrow \text{wf_tuple } n (\bigcup A \in \text{set } A_pos. A) z \wedge$
 $\text{list_all2 } (\lambda A X. \text{restrict } A z \in X) A_pos (\text{take } (\text{length } A_pos) L) \wedge$
 $\text{list_all2 } (\lambda A X. \text{restrict } A z \notin X) A_neg (\text{drop } (\text{length } A_pos) L)$
proof -
define *Q* **where** *Q* = *set (zip A_pos L)*
have *Q_alt*: *Q* = *set (zip A_pos (take (length A_pos) L))*
unfolding *Q_def* **by** (*fastforce simp: in_set_zip*)
define *Q_neg* **where** *Q_neg* = *set (zip A_neg (drop (length A_pos) L))*
let *?r* = *mmulti_join' A_pos A_neg L*
have *?r* = *New_max_getIJ_wrapperGenericJoin Q Q_neg*

```

    by (simp add: Q_def Q_neg_def)
  moreover have card Q ≥ 1
    unfolding Q_def using assms(1,2)
    by (auto simp: Suc_le_eq card_gt_0_iff zip_eq_Nil_iff)
  moreover have ∀(A, X)∈(Q ∪ Q_neg). table n A X ∧ wf_set n A
    unfolding Q_alt Q_neg_def using assms(2) by (simp add: zip_append1 list_all2_iff)
  ultimately have z ∈ ?r ↔ wf_tuple n (⋃(A, X)∈Q. A) z ∧
    (∀(A, X)∈Q. restrict A z ∈ X) ∧ (∀(A, X)∈Q_neg. restrict A z ∉ X)
    using New_max.wrapper_correctness case_prod_beta' by blast
  moreover have (⋃ A∈set A_pos. A) = (⋃(A, X)∈Q. A) proof -
    from assms(2) have length A_pos ≤ length L by (auto dest!: list_all2_lengthD)
  then show ?thesis
    unfolding Q_alt
    by (auto elim: in_set_impl_in_set_zip1[rotated, where ys=take (length A_pos) L]
      dest: set_zip_leftD)
  qed
  moreover have ∧z. (∀(A, X)∈Q. restrict A z ∈ X) ↔
    list_all2 (λA X. restrict A z ∈ X) A_pos (take (length A_pos) L)
    unfolding Q_alt using assms(2) by (auto simp add: list_all2_iff)
  moreover have ∧z. (∀(A, X)∈Q_neg. restrict A z ∉ X) ↔
    list_all2 (λA X. restrict A z ∉ X) A_neg (drop (length A_pos) L)
    unfolding Q_neg_def using assms(2) by (auto simp add: list_all2_iff)
  ultimately show ?thesis
    unfolding Q_def Q_neg_def using assms(2) by simp
  qed

lemmas restrict_nested = New_max.restrict_nested

lemma list_all2_opt_True:
  assumes list_all2 (λA X. table n A X ∧ 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 (λA X. table n A X ∧ wf_set n A)
  ((A_zs @ (A_x ∪ A_y) # A_xs @ A_ys) @ A_neg) ((zs @ join x True y # xs @ ys) @ L_neg)
  proof -
  have assms_dest: table n A_x x table n A_y y wf_set n A_x wf_set n A_y
  using assms
  by (auto simp del: mmulti_join'.simps simp add: list_all2_append1 dest: list_all2_lengthD)
  then have tabs: table n (A_x ∪ A_y) (join x True y) wf_set n (A_x ∪ A_y)
  using join_table[of n A_x x A_y y True A_x ∪ A_y, OF assms_dest(1,2)] assms_dest(3,4)
  by (auto simp add: wf_set_def)
  then show list_all2 (λA X. table n A X ∧ wf_set n A)
  ((A_zs @ (A_x ∪ A_y) # A_xs @ A_ys) @ A_neg) ((zs @ join x True y # xs @ ys) @ L_neg)
  using assms
  by (auto simp del: mmulti_join'.simps simp add: list_all2_append1 list_all2_append2
    list_all2_Cons1 list_all2_Cons2 dest: list_all2_lengthD) fastforce
  qed

lemma mmulti_join'_opt_True:
  assumes list_all2 (λA X. table n A X ∧ 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 ∪ A_y) # A_xs @ A_ys) A_neg
  ((zs @ join x True y # xs @ ys) @ L_neg)

```

proof –

```

have assms_dest: table n A_x x table n A_y y wf_set n A_x wf_set n A_y
  using assms
  by (auto simp del: mmulti_join'.simps simp add: list_all2_append1 dest: list_all2_lengthD)
then have tabs: table n (A_x  $\cup$  A_y) (join x True y) wf_set n (A_x  $\cup$  A_y)
  using join_table[of n A_x x A_y y True A_x  $\cup$  A_y, OF assms_dest(1,2)] assms_dest(3,4)
  by (auto simp add: wf_set_def)
then have list_all2': list_all2 ( $\lambda A X. \text{table } n A X \wedge \text{wf\_set } n A$ )
  ((A_zs  $\@$  (A_x  $\cup$  A_y)  $\#$  A_xs  $\@$  A_ys)  $\@$  A_neg) ((zs  $\@$  join x True y  $\#$  xs  $\@$  ys)  $\@$  L_neg)
  using assms
  by (auto simp del: mmulti_join'.simps simp add: list_all2_append1 list_all2_append2
    list_all2_Cons1 list_all2_Cons2 dest: list_all2_lengthD) fastforce
have res:  $\bigwedge z Z. \text{wf\_tuple } n Z z \implies A_x \cup A_y \subseteq Z \implies$ 
  restrict (A_x  $\cup$  A_y) z  $\in$  join x True y  $\iff$  restrict A_x z  $\in$  x  $\wedge$  restrict A_y z  $\in$  y
  using join_restrict[of x n A_x y A_y True] wf_tuple_restrict_simple[of n  $\_ \_$  A_x  $\cup$  A_y]
    assms_dest(1,2)
  by (auto simp add: table_def restrict_nested Int_absorb2)
show ?thesis
proof (rule set_eqI, rule iffI)
  fix z
  assume z  $\in$  mmulti_join' (A_zs  $\@$  A_x  $\#$  A_xs  $\@$  A_y  $\#$  A_ys) A_neg
    ((zs  $\@$  x  $\#$  xs  $\@$  y  $\#$  ys)  $\@$  L_neg)
  then have z_in_dest: wf_tuple n ( $\bigcup$  (set (A_zs  $\@$  A_x  $\#$  A_xs  $\@$  A_y  $\#$  A_ys))) z
    list_all2 ( $\lambda A. (\in) (\text{restrict } A z) A_zs zs$ )
    restrict A_x z  $\in$  x
    list_all2 ( $\lambda A. (\in) (\text{restrict } A z) A_ys ys$ )
    restrict A_y z  $\in$  y
    list_all2 ( $\lambda A. (\in) (\text{restrict } A z) A_xs xs$ )
    list_all2 ( $\lambda A. (\notin) (\text{restrict } A z) A_neg L_neg$ )
    using mmulti_join'_correct[OF  $\_$  assms(1), of z]
    by (auto simp del: mmulti_join'.simps simp add: assms list_all2_append1
      dest: list_all2_lengthD)
  then show z  $\in$  mmulti_join' (A_zs  $\@$  (A_x  $\cup$  A_y)  $\#$  A_xs  $\@$  A_ys) A_neg
    ((zs  $\@$  join x True y  $\#$  xs  $\@$  ys)  $\@$  L_neg)
    using mmulti_join'_correct[OF  $\_$  list_all2', of z] res[OF z_in_dest(1)]
    by (auto simp add: assms list_all2_appendI le_supI2 Un_assoc simp del: mmulti_join'.simps
      dest: list_all2_lengthD)

```

next

fix *z*

```

assume z  $\in$  mmulti_join' (A_zs  $\@$  (A_x  $\cup$  A_y)  $\#$  A_xs  $\@$  A_ys) A_neg
  ((zs  $\@$  join x True y  $\#$  xs  $\@$  ys)  $\@$  L_neg)
then have z_in_dest: wf_tuple n ( $\bigcup$  (set (A_zs  $\@$  A_x  $\#$  A_xs  $\@$  A_y  $\#$  A_ys))) z
  list_all2 ( $\lambda A. (\in) (\text{restrict } A z) A_zs zs$ )
  restrict (A_x  $\cup$  A_y) z  $\in$  join x True y
  list_all2 ( $\lambda A. (\in) (\text{restrict } A z) A_ys ys$ )
  list_all2 ( $\lambda A. (\in) (\text{restrict } A z) A_xs xs$ )
  list_all2 ( $\lambda A. (\notin) (\text{restrict } A z) A_neg L_neg$ )
  using mmulti_join'_correct[OF  $\_$  list_all2', of z]
  by (auto simp del: mmulti_join'.simps simp add: assms list_all2_append Un_assoc
    dest: list_all2_lengthD)
then show z  $\in$  mmulti_join' (A_zs  $\@$  A_x  $\#$  A_xs  $\@$  A_y  $\#$  A_ys) A_neg
  ((zs  $\@$  x  $\#$  xs  $\@$  y  $\#$  ys)  $\@$  L_neg)
  using mmulti_join'_correct[OF  $\_$  assms(1), of z] res[OF z_in_dest(1)]
  by (auto simp add: assms list_all2_appendI le_supI2 Un_assoc simp del: mmulti_join'.simps
    dest: list_all2_lengthD)

```

qed

qed

lemma *list_all2_opt_False*:

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

proof –

have *assms_dest*: $\text{table } n A_x x \text{ table } n A_y y \text{ wf_set } n A_x \text{ wf_set } n A_y$
using *assms*
by (*auto simp del: mmulti_join'.simps simp add: list_all2_append dest: list_all2_lengthD*)
have *tabs*: $\text{table } n A_x (\text{join } x \text{ False } y)$
using *join_table*[*of* $n A_x x A_y y \text{ False } A_x$, *OF* *assms_dest*(1,2) *assms*(6)] *assms*(6) **by** *auto*
then show *list_all2* ($\lambda A X. \text{table } n A X \wedge \text{wf_set } n A$)
 $((A_zs @ A_x \# A_xs) @ (A_ws @ A_y \# A_ys)) ((zs @ \text{join } x \text{ False } y \# xs) @ (ws @ ys))$
using *assms assms_dest*(3)
by (*auto simp del: mmulti_join'.simps simp add: list_all2_append1 list_all2_append2*
list_all2_Cons1 list_all2_Cons2 dest: list_all2_lengthD) *fastforce*

qed

lemma *mmulti_join'_opt_False*:

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

proof –

have *assms_dest*: $\text{table } n A_x x \text{ table } n A_y y \text{ wf_set } n A_x \text{ wf_set } n A_y$
using *assms*
by (*auto simp del: mmulti_join'.simps simp add: list_all2_append dest: list_all2_lengthD*)
have *tabs*: $\text{table } n A_x (\text{join } x \text{ False } y)$
using *join_table*[*of* $n A_x x A_y y \text{ False } A_x$, *OF* *assms_dest*(1,2) *assms*(6)] *assms*(6) **by** *auto*
then have *list_all2'*: *list_all2* ($\lambda A X. \text{table } n A X \wedge \text{wf_set } n A$)
 $((A_zs @ A_x \# A_xs) @ (A_ws @ A_y \# A_ys)) ((zs @ \text{join } x \text{ False } y \# xs) @ (ws @ ys))$
using *assms assms_dest*(3)
by (*auto simp del: mmulti_join'.simps simp add: list_all2_append1 list_all2_append2*
list_all2_Cons1 list_all2_Cons2 dest: list_all2_lengthD) *fastforce*
have *res*: $\bigwedge z. \text{restrict } A_x z \in \text{join } x \text{ False } y \longleftrightarrow \text{restrict } A_x z \in x \wedge \text{restrict } A_y z \notin y$
using *join_restrict*[*of* $x n A_x y A_y \text{ False}$, *OF* *assms*(6)] *assms_dest*(1,2) *assms*(6)
by (*auto simp add: table_def restrict_nested Int_absorb2 Un_absorb2*)

show *?thesis*

proof (*rule set_eqI, rule iffI*)

fix z

assume $z \in \text{mmulti_join}' (A_zs @ A_x \# A_xs) (A_ws @ A_y \# A_ys)$
 $((zs @ x \# xs) @ ws @ y \# ys)$

then have $z \text{ in_dest: wf_tuple } n (\bigcup (\text{set } (A_zs @ A_x \# A_xs))) z$

list_all2 ($\lambda A. (\in) (\text{restrict } A z) A_zs zs$)

$\text{restrict } A_x z \in x$

list_all2 ($\lambda A. (\in) (\text{restrict } A z) A_xs xs$)

list_all2 ($\lambda A. (\notin) (\text{restrict } A z) A_ws ws$)

$\text{restrict } A_y z \notin y$

list_all2 ($\lambda A. (\notin) (\text{restrict } A z) A_ys ys$)

using *mmulti_join'_correct*[*OF* *assms*(1), *of* z]

by (*auto simp del: mmulti_join'.simps simp add: assms list_all2_append1*)

```

    dest: list_all2_lengthD)
then show  $z \in \text{mmulti\_join}' (A\_zs @ A\_x \# A\_xs) (A\_ws @ A\_ys)$ 
  (( $zs @ \text{join } x \text{ False } y \# xs$ ) @  $ws @ ys$ )
using  $\text{mmulti\_join}'\_correct[OF\_list\_all2', \text{of } z]$   $res$ 
by ( $\text{auto simp add: assms list\_all2\_appendI Un\_assoc simp del: mmulti\_join}'.simps$ 
  dest: list_all2_lengthD)
next
fix  $z$ 
assume  $z \in \text{mmulti\_join}' (A\_zs @ A\_x \# A\_xs) (A\_ws @ A\_ys)$ 
  (( $zs @ \text{join } x \text{ False } y \# xs$ ) @  $ws @ ys$ )
then have  $z \text{ in\_dest: wf\_tuple } n (\bigcup (\text{set } (A\_zs @ A\_x \# A\_xs))) z$ 
  list_all2 ( $\lambda A. (\in) (\text{restrict } A z)$ )  $A\_zs zs$ 
  restrict  $A\_x z \in \text{join } x \text{ False } y$ 
  list_all2 ( $\lambda A. (\in) (\text{restrict } A z)$ )  $A\_xs xs$ 
  list_all2 ( $\lambda A. (\notin) (\text{restrict } A z)$ )  $A\_ws ws$ 
  list_all2 ( $\lambda A. (\notin) (\text{restrict } A z)$ )  $A\_ys ys$ 
using  $\text{mmulti\_join}'\_correct[OF\_list\_all2', \text{of } z]$ 
by ( $\text{auto simp del: mmulti\_join}'.simps simp add: assms list\_all2\_append1$ 
  dest: list_all2_lengthD)
then show  $z \in \text{mmulti\_join}' (A\_zs @ A\_x \# A\_xs) (A\_ws @ A\_y \# A\_ys)$ 
  (( $zs @ x \# xs$ ) @  $ws @ y \# ys$ )
using  $\text{mmulti\_join}'\_correct[OF\_assms(1), \text{of } z]$   $res$ 
by ( $\text{auto simp add: assms list\_all2\_appendI Un\_assoc simp del: mmulti\_join}'.simps$ 
  dest: list_all2_lengthD)
qed
qed

fun  $\text{find\_sub\_in} :: 'a \text{ set} \Rightarrow 'a \text{ set list} \Rightarrow \text{bool} \Rightarrow$ 
  ( $'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list}$ )  $\text{option}$  where
   $\text{find\_sub\_in } X [] b = \text{None}$ 
|  $\text{find\_sub\_in } X (x \# xs) b = (\text{if } (x \subseteq X \vee (b \wedge X \subseteq x)) \text{ then } \text{Some } ([], x, xs)$ 
   $\text{else } (\text{case } \text{find\_sub\_in } X xs b \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } (ys, z, zs) \Rightarrow \text{Some } (x \# ys, z, zs)))$ 

lemma  $\text{find\_sub\_in\_sound: find\_sub\_in } X xs b = \text{Some } (ys, z, zs) \Longrightarrow$ 
   $xs = ys @ z \# zs \wedge (z \subseteq X \vee (b \wedge X \subseteq z))$ 
by ( $\text{induction } X xs b \text{ arbitrary: } ys z zs \text{ rule: find\_sub\_in.induct}$ 
  ( $\text{fastforce split: if\_splits option.splits}$ )+)

fun  $\text{find\_sub\_True} :: 'a \text{ set list} \Rightarrow$ 
  ( $'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list}$ )  $\text{option}$  where
   $\text{find\_sub\_True} [] = \text{None}$ 
|  $\text{find\_sub\_True } (x \# xs) = (\text{case } \text{find\_sub\_in } x xs \text{ True of } \text{None} \Rightarrow$ 
  ( $\text{case } \text{find\_sub\_True } xs \text{ of } \text{None} \Rightarrow \text{None}$ 
  |  $\text{Some } (ys, w, ws, z, zs) \Rightarrow \text{Some } (x \# ys, w, ws, z, zs)$ )
  |  $\text{Some } (ys, z, zs) \Rightarrow \text{Some } ([], x, ys, z, zs)$ )

lemma  $\text{find\_sub\_True\_sound: find\_sub\_True } xs = \text{Some } (ys, w, ws, z, zs) \Longrightarrow$ 
   $xs = ys @ w \# ws @ z \# zs \wedge (z \subseteq w \vee w \subseteq z)$ 
using  $\text{find\_sub\_in\_sound}$ 
by ( $\text{induction } xs \text{ arbitrary: } ys w ws z zs \text{ rule: find\_sub\_True.induct}$ 
  ( $\text{fastforce split: option.splits}$ )+)

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

```

| *Some* (*ys*, *z*, *zs*) \Rightarrow *Some* ($([], x, xs), (ys, z, zs)$)

lemma *find_sub_False_sound*: *find_sub_False* *xs ns* = *Some* ($(rs, w, ws), (ys, z, zs)$) \Rightarrow
xs = *rs* @ *w* # *ws* \wedge *ns* = *ys* @ *z* # *zs* \wedge (*z* \subseteq *w*)
using *find_sub_in_sound*
by (*induction xs ns arbitrary: rs w ws ys z zs rule: find_sub_False.induct*)
(fastforce split: option.splits)+

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

lemma *proj_list_3_same*:
assumes *proj_list_3 xs (ys, z, zs)* = (*ys'*, *z'*, *zs'*)
length xs = *length ys* + 1 + *length zs*
shows *xs* = *ys'* @ *z'* # *zs'*
using *assms* **by** (*auto simp add: id_take_nth_drop*)

lemma *proj_list_3_length*:
assumes *proj_list_3 xs (ys, z, zs)* = (*ys'*, *z'*, *zs'*)
length xs = *length ys* + 1 + *length zs*
shows *length ys* = *length ys'* *length zs* = *length zs'*
using *assms* **by** *auto*

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

lemma *proj_list_5_same*:
assumes *proj_list_5 xs (ys, w, ws, z, zs)* = (*ys'*, *w'*, *ws'*, *z'*, *zs'*)
length xs = *length ys* + 1 + *length ws* + 1 + *length zs*
shows *xs* = *ys'* @ *w'* # *ws'* @ *z'* # *zs'*

proof –

have *xs* ! *length ys* # *take* (*length ws*) (*drop* (*Suc* (*length ys*)) *xs*) = *take* (*Suc* (*length ws*)) (*drop* (*length ys*) *xs*)

using *assms(2)* **by** (*simp add: list_eq_iff_nth_eq_nth_Cons split: nat.split*)

moreover **have** *take* (*Suc* (*length ws*)) (*drop* (*length ys*) *xs*) @ *drop* (*Suc* (*length ys* + *length ws*)) *xs* =
drop (*length ys*) *xs*

unfolding *Suc_eq_plus1 add.assoc[of _ 1] add.commute[of _ length ws + 1]*

drop_drop[symmetric, of length ws + 1] append_take_drop_id ..

ultimately **show** *?thesis*

using *assms* **by** (*auto simp: Cons_nth_drop_Suc append_Cons[symmetric]*)

qed

lemma *proj_list_5_length*:
assumes *proj_list_5 xs (ys, w, ws, z, zs)* = (*ys'*, *w'*, *ws'*, *z'*, *zs'*)
length xs = *length ys* + 1 + *length ws* + 1 + *length zs*
shows *length ys* = *length ys'* *length ws* = *length ws'*
length zs = *length zs'*
using *assms* **by** *auto*

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* $xs = \text{Some } (ys, w, ws, z, zs)$ $\text{length } xs = \text{length } xs'$

proj_list_5 $xs' (ys, w, ws, z, zs) = (ys', w', ws', z', zs')$

shows $xs' = ys' @ w' \# ws' @ z' \# zs'$

proof –

have *len*: $\text{length } xs' = \text{length } ys + 1 + \text{length } ws + 1 + \text{length } zs$

using *find_sub_True_sound*[*OF* *assms*(1)] **by** (*auto simp add: assms*(2)[*symmetric*])

show *?thesis*

using *proj_list_5_same*[*OF* *assms*(3) *len*] .

qed

lemma *find_sub_True_proj_list_5_length*:

assumes *find_sub_True* $xs = \text{Some } (ys, w, ws, z, zs)$ $\text{length } xs = \text{length } xs'$

proj_list_5 $xs' (ys, w, ws, z, zs) = (ys', w', ws', z', zs')$

shows $\text{length } ys = \text{length } ys'$ $\text{length } ws = \text{length } ws'$

$\text{length } zs = \text{length } zs'$

using *find_sub_True_sound*[*OF* *assms*(1)] *proj_list_5_length*[*OF* *assms*(3)] *assms*(2) **by** *auto*

lemma *dominate_True_sound*:

assumes *dominate_True* $A_pos L_pos = \text{Some } ((A_zs, A_x, A_xs, A_y, A_ys), (zs, x, xs, y, ys))$

$\text{length } A_pos = \text{length } L_pos$

shows $A_pos = A_zs @ A_x \# A_xs @ A_y \# A_ys$ $L_pos = zs @ x \# xs @ y \# ys$

$\text{length } A_xs = \text{length } xs$ $\text{length } A_ys = \text{length } ys$ $\text{length } A_zs = \text{length } zs$

using *assms* *find_sub_True_sound* *find_sub_True_proj_list_5_same* *find_sub_True_proj_list_5_length*

by (*auto simp del: proj_list_5.simps split: option.splits*) *fast+*

fun *dominate_False* :: *nat set list* \Rightarrow *'a table list* \Rightarrow *nat set list* \Rightarrow *'a table list* \Rightarrow

$((((\text{nat set list} \times \text{nat set} \times \text{nat set list}) \times \text{nat set list} \times \text{nat set} \times \text{nat set list}) \times$

$(('a \text{ table list} \times 'a \text{ table} \times 'a \text{ table list}) \times$

$'a \text{ table list} \times 'a \text{ table} \times 'a \text{ table list}))$ *option where*

dominate_False $A_pos L_pos A_neg L_neg = (\text{case } \text{find_sub_False } A_pos A_neg \text{ of } \text{None} \Rightarrow \text{None}$

| *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* $xs ns = \text{Some } ((rs, w, ws), (ys, z, zs))$

$\text{length } xs = \text{length } xs'$ *proj_list_3* $xs' (rs, w, ws) = (rs', w', ws')$

shows $xs' = rs' @ w' \# ws'$

proof –

have *len*: $\text{length } xs' = \text{length } rs + 1 + \text{length } ws$

using *find_sub_False_sound*[*OF* *assms*(1)] **by** (*auto simp add: assms*(2)[*symmetric*])

show *?thesis*

using *proj_list_3_same*[*OF* *assms*(3) *len*] .

qed

lemma *find_sub_False_proj_list_3_length_left*:

assumes *find_sub_False* $xs ns = \text{Some } ((rs, w, ws), (ys, z, zs))$

$\text{length } xs = \text{length } xs'$ *proj_list_3* $xs' (rs, w, ws) = (rs', w', ws')$

shows $\text{length } rs = \text{length } rs'$ $\text{length } ws = \text{length } ws'$

using *find_sub_False_sound*[*OF* *assms*(1)] *proj_list_3_length*[*OF* *assms*(3)] *assms*(2) **by** *auto*

lemma *find_sub_False_proj_list_3_same_right*:

assumes *find_sub_False* $xs ns = \text{Some } ((rs, w, ws), (ys, z, zs))$

$\text{length } ns = \text{length } ns'$ *proj_list_3* $ns' (ys, z, zs) = (ys', z', zs')$

shows $ns' = ys' @ z' \# zs'$

proof –

```

have len: length ns' = length ys + 1 + length zs
  using find_sub_False_sound[OF assms(1)] by (auto simp add: assms(2)[symmetric])
show ?thesis
  using proj_list_3_same[OF assms(3) len] .
qed

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'
using find_sub_False_sound[OF assms(1)] proj_list_3_length[OF assms(3)] assms(2) by auto

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 ⊆ A_x
using assms find_sub_False_proj_list_3_same_left find_sub_False_proj_list_3_same_right
  find_sub_False_proj_list_3_length_left find_sub_False_proj_list_3_length_right
  find_sub_False_sound
by (auto simp del: proj_list_3.simps split: option.splits) fast+

function mmulti_join :: (nat ⇒ nat set list ⇒ nat set list ⇒ 'a table list ⇒ 'a table) where
  mmulti_join n A_pos A_neg L = (if length A_pos + length A_neg ≠ 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 ⇒
      (case dominate_False A_pos L_pos A_neg L_neg of None ⇒ 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)) ⇒
          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)) ⇒
          mmulti_join n (A_zs @ (A_x ∪ A_y) # A_xs @ A_ys) A_neg
            ((zs @ bin_join n A_x x True A_y y # xs @ ys) @ L_neg)))
    by pat_completeness auto

termination
by (relation measure (λ(n, A_pos, A_neg, L). length A_pos + length A_neg))
  (use find_sub_True_sound find_sub_False_sound in ‹fastforce split: option.splits›)+

lemma mmulti_join_link:
assumes A_pos ≠ []
  and list_all2 (λA X. table n A X ∧ 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
using assms
proof (induction A_pos A_neg L rule: mmulti_join.induct)
case (1 n A_pos A_neg L)
define L_pos where L_pos = take (length A_pos) L
define L_neg where L_neg = drop (length A_pos) L
have L_def: L = L_pos @ L_neg
  using L_pos_def L_neg_def by auto
have lens_match: length A_pos = length L_pos length A_neg = length L_neg
  using L_pos_def L_neg_def 1(4)[unfolded L_def] by (auto dest: list_all2_lengthD)
then have lens_sum: length A_pos + length A_neg = length L
  by (auto simp add: L_def)
show ?case

```

```

proof (cases dominate_True A_pos L_pos)
  case None
  note dom_True = None
  show ?thesis
  proof (cases dominate_False A_pos L_pos A_neg L_neg)
    case None
    show ?thesis
    by (subst mmulti_join.simps)
      (simp del: dominate_True.simps dominate_False.simps mmulti_join.simps
        mmulti_join'.simps add: Let_def dom_True L_pos_def[symmetric] None
        L_neg_def[symmetric] lens_sum split: option.splits)
  next
  case (Some a)
  then obtain A_zs A_x A_xs A_ws A_y A_ys zs x xs ws y ys where
    dom_False: 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))
  by (cases a) auto
  note list_all2 = 1(4)[unfolded L_def dominate_False_sound[OF dom_False lens_match]]
  have lens: length A_ws = length ws length A_xs = length xs
    length A_ys = length ys length A_zs = length zs
  using dominate_False_sound[OF dom_False lens_match] by auto
  have sub: A_y  $\subseteq$  A_x
  using dominate_False_sound[OF dom_False lens_match] by auto
  have list_all2': 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))
  using list_all2_opt_False[OF list_all2 lens sub] .
  have tabs: table n A_x x table n A_y y
  using list_all2 by (auto simp add: lens list_all2_append)
  have bin_join_conv: join x False y = bin_join n A_x x False A_y y
  using bin_join_table[OF tabs, symmetric] .
  have mmulti: mmulti_join n A_pos A_neg L = 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))
  by (subst mmulti_join.simps)
    (simp del: dominate_True.simps dominate_False.simps mmulti_join.simps
      add: Let_def dom_True L_pos_def[symmetric] L_neg_def[symmetric] dom_False lens_sum)
  show ?thesis
  unfolding mmulti
  unfolding L_def dominate_False_sound[OF dom_False lens_match]
  by (rule 1(1)[OF _ L_pos_def L_neg_def dom_True dom_False,
    OF _____ list_all2'[unfolded bin_join_conv],
    unfolded mmulti_join'_opt_False[OF list_all2 lens sub, symmetric,
    unfolded bin_join_conv]])
    (auto simp add: lens_sum)
  qed
next
  case (Some a)
  then obtain A_zs A_x A_xs A_y A_ys zs x xs y ys where dom_True: dominate_True A_pos
L_pos =
    Some ((A_zs, A_x, A_xs, A_y, A_ys), (zs, x, xs, y, ys))
  by (cases a) auto
  note list_all2 = 1(4)[unfolded L_def dominate_True_sound[OF dom_True lens_match(1)]]
  have lens: length A_xs = length xs length A_ys = length ys length A_zs = length zs
  using dominate_True_sound[OF dom_True lens_match(1)] by auto
  have list_all2': 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)
  using list_all2_opt_True[OF list_all2 lens] .
  have tabs: table n A_x x table n A_y y

```

```

using list_all2 by (auto simp add: lens list_all2_append)
have bin_join_conv: join x True y = bin_join n A_x x True A_y y
using bin_join_table[OF tabs, symmetric] .
have mmulti: mmulti_join n A_pos A_neg L = mmulti_join n (A_zs @ (A_x ∪ A_y) # A_xs @
A_ys)
  A_neg ((zs @ bin_join n A_x x True A_y y # xs @ ys) @ L_neg)
by (subst mmulti_join.simps)
  (simp del: dominate_True.simps dominate_False.simps mmulti_join.simps
  add: Let_def dom_True L_pos_def[symmetric] L_neg_def lens_sum)
show ?thesis
unfolding mmulti
unfolding L_def dominate_True_sound[OF dom_True lens_match(1)]
by (rule 1(2)[OF _ L_pos_def L_neg_def dom_True,
  OF _ _ _ _ _ list_all2'[unfolded bin_join_conv],
  unfolded mmulti_join'_opt_True[OF list_all2 lens, symmetric,
  unfolded bin_join_conv]])
  (auto simp add: lens_sum)
qed
qed

```

lemma mmulti_join_correct:

```

assumes A_pos ≠ []
and list_all2 (λA X. table n A X ∧ wf_set n A) (A_pos @ A_neg) L
shows z ∈ mmulti_join n A_pos A_neg L ↔ wf_tuple n (⋃ A ∈ set A_pos. A) z ∧
  list_all2 (λA X. restrict A z ∈ X) A_pos (take (length A_pos) L) ∧
  list_all2 (λA X. restrict A z ∉ X) A_neg (drop (length A_pos) L)
unfolding mmulti_join_link[OF assms] using mmulti_join'_correct[OF assms] .

```

6 Generic monitoring algorithm

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

6.1 Monitorable formulas

definition mmonitorable $\varphi \longleftrightarrow \text{safe_formula } \varphi \wedge \text{Formula.future_bounded } \varphi$

definition mmonitorable_regex $b \ g \ r \longleftrightarrow \text{safe_regex } b \ g \ r \wedge \text{Regex.pred_regex } \text{Formula.future_bounded } r$

definition is_simple_eq :: $\text{Formula.trm} \Rightarrow \text{Formula.trm} \Rightarrow \text{bool}$ **where**

```

is_simple_eq t1 t2 = (Formula.is_Const t1 ∧ (Formula.is_Const t2 ∨ Formula.is_Var t2) ∨
  Formula.is_Var t1 ∧ Formula.is_Const t2)

```

fun mmonitorable_exec :: $\text{Formula.formula} \Rightarrow \text{bool}$ **where**

```

mmonitorable_exec (Formula.Eq t1 t2) = is_simple_eq t1 t2
| mmonitorable_exec (Formula.Neg (Formula.Eq (Formula.Var x) (Formula.Var y))) = (x = y)
| mmonitorable_exec (Formula.Pred e ts) = list_all (λt. Formula.is_Var t ∨ Formula.is_Const t) ts
| mmonitorable_exec (Formula.Let p  $\varphi$   $\psi$ ) = ({0..<Formula.nfv  $\varphi$ } ⊆ Formula.fv  $\varphi$  ∧ mmonitorable_exec  $\varphi$ 
∧ mmonitorable_exec  $\psi$ )
| mmonitorable_exec (Formula.Neg  $\varphi$ ) = (fv  $\varphi$  = {} ∧ mmonitorable_exec  $\varphi$ )
| mmonitorable_exec (Formula.Or  $\varphi$   $\psi$ ) = (fv  $\varphi$  = fv  $\psi$  ∧ mmonitorable_exec  $\varphi$  ∧ mmonitorable_exec  $\psi$ )
| mmonitorable_exec (Formula.And  $\varphi$   $\psi$ ) = (mmonitorable_exec  $\varphi$  ∧
  (safe_assignment (fv  $\varphi$ )  $\psi$  ∨ mmonitorable_exec  $\psi$  ∨
  fv  $\psi$  ⊆ fv  $\varphi$  ∧ (is_constraint  $\psi$  ∨ (case  $\psi$  of Formula.Neg  $\psi'$  ⇒ mmonitorable_exec  $\psi'$  | _ ⇒
  False))))
| mmonitorable_exec (Formula.Ands l) = (let (pos, neg) = partition mmonitorable_exec l in

```

```

pos ≠ [] ∧ list_all mmonitorable_exec (map remove_neg neg) ∧
  ⋃(set (map fv neg)) ⊆ ⋃(set (map fv pos)))
| mmonitorable_exec (Formula.Exists φ) = (mmonitorable_exec φ)
| mmonitorable_exec (Formula.Agg y ω b f φ) = (mmonitorable_exec φ ∧
  y + b ∉ Formula.fv φ ∧ {0..<b} ⊆ Formula.fv φ ∧ Formula.fv_trm f ⊆ Formula.fv φ)
| mmonitorable_exec (Formula.Prev I φ) = (mmonitorable_exec φ)
| mmonitorable_exec (Formula.Next I φ) = (mmonitorable_exec φ)
| mmonitorable_exec (Formula.Since φ I ψ) = (Formula.fv φ ⊆ Formula.fv ψ ∧
  (mmonitorable_exec φ ∨ (case φ of Formula.Neg φ' ⇒ mmonitorable_exec φ' | _ ⇒ False))) ∧
  mmonitorable_exec ψ)
| mmonitorable_exec (Formula.Until φ I ψ) = (Formula.fv φ ⊆ Formula.fv ψ ∧ right I ≠ ∞ ∧
  (mmonitorable_exec φ ∨ (case φ of Formula.Neg φ' ⇒ mmonitorable_exec φ' | _ ⇒ False))) ∧
  mmonitorable_exec ψ)
| mmonitorable_exec (Formula.MatchP I r) = (Regex.safe_regex Formula.fv (λg φ. mmonitorable_exec φ
  ∨ (g = Lax ∧ (case φ of Formula.Neg φ' ⇒ mmonitorable_exec φ' | _ ⇒ False)))) Past Strict r
| mmonitorable_exec (Formula.MatchF I r) = (Regex.safe_regex Formula.fv (λg φ. mmonitorable_exec
  φ ∨ (g = Lax ∧ (case φ of Formula.Neg φ' ⇒ mmonitorable_exec φ' | _ ⇒ False)))) Futu Strict r ∧ right
  I ≠ ∞)
| mmonitorable_exec _ = False

```

lemma cases_Neg_iff:

```

(case φ of formula.Neg ψ ⇒ P ψ | _ ⇒ False) ↔ (∃ ψ. φ = formula.Neg ψ ∧ P ψ)
by (cases φ) auto

```

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

proof (induct φ rule: safe_formula.induct)

case (8 φ ψ)

then show ?case

unfolding safe_formula.simps future_bounded.simps mmonitorable_exec.simps

by (auto simp: cases_Neg_iff)

next

case (9 φ ψ)

then show ?case

unfolding safe_formula.simps future_bounded.simps mmonitorable_exec.simps

by (auto simp: cases_Neg_iff)

next

case (10 l)

from 10.prem1(2) have bounded: Formula.future_bounded φ if φ ∈ set l for φ

using that by (auto simp: list.pred_set)

obtain poss negs where posnegs: (poss, negs) = partition safe_formula l by simp

obtain posm negm where posnegm: (posm, negm) = partition mmonitorable_exec l by simp

have set poss ⊆ set posm

proof (rule subsetI)

fix x assume x ∈ set poss

then have x ∈ set l safe_formula x using posnegs by simp_all

then have mmonitorable_exec x using 10.hyps(1) bounded by blast

then show x ∈ set posm using ⟨x ∈ set poss⟩ posnegm posnegs by simp

qed

then have set negm ⊆ set negs using posnegm posnegs by auto

obtain poss ≠ [] list_all safe_formula (map remove_neg negs)

(⋃ x ∈ set negs. fv x) ⊆ (⋃ x ∈ set poss. fv x)

using 10.prem1(1) posnegs by simp

then have posm ≠ [] using ⟨set poss ⊆ set posm⟩ by auto

moreover have list_all mmonitorable_exec (map remove_neg negm)

proof –

let ?l = map remove_neg negm

have ∧ x. x ∈ set ?l ⇒ mmonitorable_exec x

```

proof –
  fix  $x$  assume  $x \in \text{set } ?l$ 
  then obtain  $y$  where  $y \in \text{set } \text{negm } x = \text{remove\_neg } y$  by auto
  then have  $y \in \text{set } \text{negs}$  using  $\langle \text{set } \text{negm} \subseteq \text{set } \text{negs} \rangle$  by blast
  then have safe_formula  $x$ 
    unfolding  $\langle x = \text{remove\_neg } y \rangle$  using  $\langle \text{list\_all } \text{safe\_formula } (\text{map } \text{remove\_neg } \text{negs}) \rangle$ 
    by (simp add: list_all_def)
  show mmonitorable_exec  $x$ 
  proof (cases  $\exists z. y = \text{Formula.Neg } z$ )
    case True
      then obtain  $z$  where  $y = \text{Formula.Neg } z$  by blast
      then show ?thesis
        using 10.hyps(2)[OF posnegs refl]  $\langle x = \text{remove\_neg } y \rangle \langle y \in \text{set } \text{negs} \rangle$  posnegs bounded
         $\langle \text{safe\_formula } x \rangle$  by fastforce
    next
      case False
      then have  $\text{remove\_neg } y = y$  by (cases  $y$ ) simp_all
      then have  $y = x$  unfolding  $\langle x = \text{remove\_neg } y \rangle$  by simp
      show ?thesis
        using 10.hyps(1)  $\langle y \in \text{set } \text{negs} \rangle$  posnegs  $\langle \text{safe\_formula } x \rangle$  unfolding  $\langle y = x \rangle$ 
        by auto
      qed
    qed
  then show ?thesis by (simp add: list_all_iff)
qed
moreover have  $(\bigcup_{x \in \text{set } \text{negm}. \text{fv } x} \subseteq \bigcup_{x \in \text{set } \text{posm}. \text{fv } x})$ 
  using  $\langle \bigcup ( \text{fv } ' \text{set } \text{negs} ) \subseteq \bigcup ( \text{fv } ' \text{set } \text{posm} ) \rangle \langle \text{set } \text{posm} \subseteq \text{set } \text{posm} \rangle \langle \text{set } \text{negm} \subseteq \text{set } \text{negs} \rangle$ 
  by fastforce
ultimately show ?case using posnegm by simp
next
case (15  $\varphi I \psi$ )
then show ?case
  unfolding safe_formula.simps future_bounded.simps mmonitorable_exec.simps
  by (auto simp: cases_Neg_iff)
next
case (16  $\varphi I \psi$ )
then show ?case
  unfolding safe_formula.simps future_bounded.simps mmonitorable_exec.simps
  by (auto simp: cases_Neg_iff)
next
case (17  $I r$ )
then show ?case
  by (auto elim!: safe_regex_mono[rotated] simp: cases_Neg_iff regex.pred_set)
next
case (18  $I r$ )
then show ?case
  by (auto elim!: safe_regex_mono[rotated] simp: cases_Neg_iff regex.pred_set)
qed (auto simp add: is_simple_eq_def list.pred_set)

lemma safe_assignment_future_bounded: safe_assignment  $X \varphi \implies \text{Formula.future\_bounded } \varphi$ 
  unfolding safe_assignment_def by (auto split: formula.splits)

lemma is_constraint_future_bounded: is_constraint  $\varphi \implies \text{Formula.future\_bounded } \varphi$ 
  by (induction rule: is_constraint.induct) simp_all

lemma mmonitorable_exec_mmonitorable: mmonitorable_exec  $\varphi \implies \text{mmonitorable } \varphi$ 
proof (induct  $\varphi$  rule: mmonitorable_exec.induct)
  case (7  $\varphi \psi$ )

```

```

then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (auto simp: cases_Neg_iff intro: safe_assignment_future_bounded is_constraint_future_bounded)
next
case (8 l)
obtain poss negs where posnegs: (poss, negs) = partition safe_formula l by simp
obtain posm negm where posnegm: (posm, negm) = partition mmonitorable_exec l by simp
have pos_monitorable: list_all mmonitorable_exec posm using posnegm by (simp add: list_all_iff)
have neg_monitorable: list_all mmonitorable_exec (map remove_neg negm)
  using 8.prem1 posnegm by (simp add: list_all_iff)
have set posm  $\subseteq$  set poss
  using 8.hyps(1) posnegs posnegm unfolding mmonitorable_def by auto
then have set negs  $\subseteq$  set negm
  using posnegs posnegm by auto

have poss  $\neq$  [] using 8.prem1 posnegm  $\langle$ set posm  $\subseteq$  set poss $\rangle$  by auto
moreover have list_all safe_formula (map remove_neg negs)
  using neg_monitorable 8.hyps(2)[OF posnegm refl]  $\langle$ set negs  $\subseteq$  set negm $\rangle$ 
  unfolding mmonitorable_def by (auto simp: list_all_iff)
moreover have  $\bigcup$  (fv ' set negm)  $\subseteq$   $\bigcup$  (fv ' set posm)
  using 8.prem1 posnegm by simp
then have  $\bigcup$  (fv ' set negs)  $\subseteq$   $\bigcup$  (fv ' set poss)
  using  $\langle$ set posm  $\subseteq$  set poss $\rangle$   $\langle$ set negs  $\subseteq$  set negm $\rangle$  by fastforce
ultimately have safe_formula (Formula.Ands l) using posnegs by simp
moreover have Formula.future_bounded (Formula.Ands l)
proof -
  have list_all Formula.future_bounded posm
    using pos_monitorable 8.hyps(1) posnegm unfolding mmonitorable_def
    by (auto elim!: list.pred_mono_strong)
  moreover have fnegm: list_all Formula.future_bounded (map remove_neg negm)
    using neg_monitorable 8.hyps(2) posnegm unfolding mmonitorable_def
    by (auto elim!: list.pred_mono_strong)
  then have list_all Formula.future_bounded negm
proof -
  have  $\bigwedge x. x \in$  set negm  $\implies$  Formula.future_bounded x
proof -
  fix x assume x  $\in$  set negm
  have Formula.future_bounded (remove_neg x) using fnegm  $\langle$ x  $\in$  set negm $\rangle$  by (simp add:
list_all_iff)
  then show Formula.future_bounded x by (cases x) auto
qed
  then show ?thesis by (simp add: list_all_iff)
qed
  ultimately have list_all Formula.future_bounded l using posnegm by (auto simp: list_all_iff)
  then show ?thesis by (auto simp: list_all_iff)
qed
ultimately show ?case unfolding mmonitorable_def ..
next
case (13  $\varphi$  I  $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce simp: cases_Neg_iff)
next
case (14  $\varphi$  I  $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce simp: one_enat_def cases_Neg_iff)
next

```

```

case (15 I r)
then show ?case
  by (auto elim!: safe_regex_mono[rotated] dest: safe_regex_safe[rotated]
    simp: mmonitorable_regex_def mmonitorable_def cases_Neg_iff regex.pred_set)
next
case (16 I r)
then show ?case
  by (auto 0 3 elim!: safe_regex_mono[rotated] dest: safe_regex_safe[rotated]
    simp: mmonitorable_regex_def mmonitorable_def cases_Neg_iff regex.pred_set)
qed (auto simp add: mmonitorable_def mmonitorable_regex_def is_simple_eq_def one_enat_def list.pred_set)

lemma monitorable_formula_code[code]: mmonitorable  $\varphi$  = mmonitorable_exec  $\varphi$ 
using mmonitorable_exec_mmonitorable safe_formula_mmonitorable_exec mmonitorable_def
by blast

```

6.2 Handling regular expressions

```

datatype mregex =
  MSkip nat
  | MTestPos nat
  | MTestNeg 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  $\wedge$  ok m s)
| ok m (MTimes r s) = (ok m r  $\wedge$  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)  $\varphi$ s = Regex.Test ( $\varphi$ s ! n)
| from_mregex (MTestNeg n)  $\varphi$ s = (if safe_formula (Formula.Neg ( $\varphi$ s ! n))
  then Regex.Test (Formula.Neg (Formula.Neg (Formula.Neg ( $\varphi$ s ! n))))
  else Regex.Test (Formula.Neg ( $\varphi$ s ! n)))
| from_mregex (MPlus r s)  $\varphi$ s = Regex.Plus (from_mregex r  $\varphi$ s) (from_mregex s  $\varphi$ s)
| from_mregex (MTimes r s)  $\varphi$ s = Regex.Times (from_mregex r  $\varphi$ s) (from_mregex s  $\varphi$ s)
| from_mregex (MStar r)  $\varphi$ s = Regex.Star (from_mregex r  $\varphi$ s)

```

primrec to_mregex_exec where

```

  to_mregex_exec (Regex.Skip n) xs = (MSkip n, xs)
| to_mregex_exec (Regex.Test  $\varphi$ ) xs = (if safe_formula  $\varphi$  then (MTestPos (length xs), xs @ [ $\varphi$ ])
  else case  $\varphi$  of Formula.Neg  $\varphi'$   $\Rightarrow$  (MTestNeg (length xs), xs @ [ $\varphi'$ ]) | _  $\Rightarrow$  (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  $\varphi$ ) = (if safe_formula  $\varphi$  then (MTestPos 0, [ $\varphi$ ])
  else case  $\varphi$  of Formula.Neg  $\varphi'$   $\Rightarrow$  (MTestNeg 0, [ $\varphi'$ ]) | _  $\Rightarrow$  (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*: $\text{shift } r \ 0 = r$
by (induct r) auto

lemma *shift_shift*: $\text{shift } (\text{shift } r \ k) \ j = \text{shift } r \ (k + j)$
by (induct r) auto

lemma *to_mregex_to_mregex_exec*:

```

case to_mregex r of (mr,  $\varphi$ s)  $\Rightarrow$  to_mregex_exec r xs = (shift mr (length xs), xs @  $\varphi$ s)
by (induct r arbitrary: xs)
(auto simp: shift_shift ac_simps split: formula.splits prod.splits)

```

lemma *to_mregex_to_mregex_exec_Nil*[code]: $\text{to_mregex } r = \text{to_mregex_exec } r \ []$
using *to_mregex_to_mregex_exec*[**where** $xs=[]$ **and** $r = r$] **by** (auto simp: *shift_0*)

lemma *ok_mono*: $ok \ m \ mr \Longrightarrow m \leq n \Longrightarrow ok \ n \ mr$
by (induct mr) auto

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

lemma *not_Neg_cases*:

```

( $\forall \psi. \varphi \neq \text{Formula.Neg } \psi$ )  $\Longrightarrow$  (case  $\varphi$  of formula.Neg  $\psi \Rightarrow f \ \psi$  | _  $\Rightarrow x$ ) = x
by (cases  $\varphi$ ) auto

```

lemma *to_mregex_exec_ok*:

```

to_mregex_exec r xs = (mr, ys)  $\Longrightarrow \exists zs. ys = xs @ zs \wedge \text{set } zs = \text{atms } r \wedge ok \ (\text{length } ys) \ mr$ 
```

proof (induct r arbitrary: xs mr ys)

case (Skip x)

then show ?case **by** (auto split: if_splits prod.splits simp: atms_def elim: ok_mono)

next

case (Test x)

show ?case **proof** (cases $\exists x'. x = \text{Formula.Neg } x'$)

case True

with Test show ?thesis **by** (auto split: if_splits prod.splits simp: atms_def elim: ok_mono)

next

case False

with Test show ?thesis **by** (auto split: if_splits prod.splits simp: atms_def not_Neg_cases elim: ok_mono)

```

qed
next
case (Plus r1 r2)
then show ?case by (fastforce split: if_splits prod.splits formula.splits simp: atms_def elim: ok_mono)
next
case (Times r1 r2)
then show ?case by (fastforce split: if_splits prod.splits formula.splits simp: atms_def elim: ok_mono)
next
case (Star r)
then show ?case by (auto split: if_splits prod.splits simp: atms_def elim: ok_mono)
qed

```

lemma *ok_shift*: $ok (i + m) (Monitor.shift r i) \longleftrightarrow ok m r$
by (*induct* *r*) *auto*

lemma *to_mregex_ok*: $to_mregex\ r = (mr, ys) \implies set\ ys = atms\ r \wedge ok\ (length\ ys)\ mr$
proof (*induct* *r* *arbitrary*: *mr* *ys*)

```

case (Skip x)
then show ?case by (auto simp: atms_def elim: ok_mono split: if_splits prod.splits)
next
case (Test x)
show ?case proof (cases  $\exists x'. x = Formula.Neg\ x'$ )
case True
with Test show ?thesis by (auto split: if_splits prod.splits simp: atms_def elim: ok_mono)
next
case False
with Test show ?thesis by (auto split: if_splits prod.splits simp: atms_def not_Neg_cases elim:
ok_mono)
qed
next
case (Plus r1 r2)
then show ?case by (fastforce simp: ok_shift atms_def elim: ok_mono split: if_splits prod.splits)
next
case (Times r1 r2)
then show ?case by (fastforce simp: ok_shift atms_def elim: ok_mono split: if_splits prod.splits)
next
case (Star r)
then show ?case by (auto simp: atms_def elim: ok_mono split: if_splits prod.splits)
qed

```

lemma *from_mregex_shift*: $from_mregex (shift\ r (length\ xs)) (xs @ ys) = from_mregex\ r\ ys$
by (*induct* *r*) (*auto* *simp*: *nth_append*)

lemma *from_mregex_to_mregex*: $safe_regex\ m\ g\ r \implies case_prod\ from_mregex (to_mregex\ r) = r$
by (*induct* *r* *rule*: *safe_regex.induct*)
(*auto* *dest*: *to_mregex_ok* *intro!*: *from_mregex_cong* *simp*: *nth_append* *from_mregex_shift* *cases_Neg_iff*
split: *prod.splits* *modality.splits*)

lemma *from_mregex_eq*: $safe_regex\ m\ g\ r \implies to_mregex\ r = (mr, \varphi s) \implies from_mregex\ mr\ \varphi s = r$
using *from_mregex_to_mregex*[*of* *m* *g* *r*] **by** *auto*

lemma *from_mregex_to_mregex_exec*: $safe_regex\ m\ g\ r \implies case_prod\ from_mregex (to_mregex_exec\ r\ xs) = r$

proof (*induct* *r* *arbitrary*: *xs* *rule*: *safe_regex.induct*)
case (*Plus* *b* *g* *r* *s*)
from *Plus*(3) *Plus*(1)[*of* *xs*] *Plus*(2)[*of* *snd* (*to_mregex_exec* *r* *xs*)] **show** ?*case*
by (*auto* *split*: *prod.splits* *simp*: *nth_append* *dest*: *to_mregex_exec_ok*
intro!: *from_mregex_cong*[**where** $m = length (snd (to_mregex_exec\ r\ xs))$])

```

next
  case (TimesF g r s)
  from TimesF(3) TimesF(2)[of xs] TimesF(1)[of snd (to_mregex_exec r xs)] show ?case
  by (auto split: prod.splits simp: nth_append dest: to_mregex_exec_ok
      intro!: from_mregex_cong[where m = length (snd (to_mregex_exec r xs))])
next
  case (TimesP g r s)
  from TimesP(3) TimesP(1)[of xs] TimesP(2)[of snd (to_mregex_exec r xs)] show ?case
  by (auto split: prod.splits simp: nth_append dest: to_mregex_exec_ok
      intro!: from_mregex_cong[where m = length (snd (to_mregex_exec r xs))])
next
  case (Star b g r)
  from Star(2) Star(1)[of xs] show ?case
  by (auto split: prod.splits)
qed (auto split: prod.splits simp: cases_Neg_iff)

```

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]:

$\text{saturate } f S = (\text{let } S' = f S \text{ in if } S' = S \text{ then } S \text{ else saturate } f S')$

unfolding saturate_def Let_def

by (subst while_unfold) auto

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

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

primrec LPD where

$\text{LPD } (\text{MSkip } n) = (\text{case } n \text{ of } 0 \Rightarrow \{\} \mid \text{Suc } m \Rightarrow \{\text{MSkip } m\})$

$\mid \text{LPD } (\text{MTestPos } \varphi) = \{\}$

$\mid \text{LPD } (\text{MTestNeg } \varphi) = \{\}$

$\mid \text{LPD } (\text{MPlus } r s) = (\text{LPD } r \cup \text{LPD } s)$

$\mid \text{LPD } (\text{MTimes } r s) = \text{MTimesR } (\text{LPD } r) s \cup \text{LPD } s$

$\mid \text{LPD } (\text{MStar } r) = \text{MTimesR } (\text{LPD } r) (\text{MStar } r)$

primrec LPDi where

$\text{LPDi } 0 r = \{r\}$

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

lemma LPDi_Suc_alt: $\text{LPDi } (\text{Suc } i) r = (\bigcup s \in \text{LPDi } i r. \text{LPD } s)$

by (induct i arbitrary: r) fastforce+

definition LPDs $r = (\bigcup i. \text{LPDi } i r)$

lemma LPDs_refl: $r \in \text{LPDs } r$

by (auto simp: LPDs_def intro: exI[of _ 0])

lemma LPDs_trans: $r \in \text{LPD } s \implies s \in \text{LPDs } t \implies r \in \text{LPDs } t$

by (force simp: LPDs_def LPDi_Suc_alt simp del: LPDi.simps intro: exI[of _ Suc _])

lemma LPDi_Test:

$\text{LPDi } i (\text{MSkip } 0) \subseteq \{\text{MSkip } 0\}$

$\text{LPDi } i (\text{MTestPos } \varphi) \subseteq \{\text{MTestPos } \varphi\}$

$\text{LPDi } i (\text{MTestNeg } \varphi) \subseteq \{\text{MTestNeg } \varphi\}$

by (induct i) auto

lemma *LPDs_Test*:
 $LPDs (MSkip\ 0) \subseteq \{MSkip\ 0\}$
 $LPDs (MTestPos\ \varphi) \subseteq \{MTestPos\ \varphi\}$
 $LPDs (MTestNeg\ \varphi) \subseteq \{MTestNeg\ \varphi\}$
unfolding *LPDs_def* **using** *LPDi_Test* **by** *blast+*

lemma *LPDi_MSkip*: $LPDi\ i\ (MSkip\ n) \subseteq MSkip\ \{i.\ i \leq n\}$
by (*induct i arbitrary: n*) (*auto dest: set_mp[OF LPDi_Test(1)] simp: le_Suc_eq split: nat.splits*)

lemma *LPDs_MSkip*: $LPDs (MSkip\ n) \subseteq MSkip\ \{i.\ i \leq n\}$
unfolding *LPDs_def* **using** *LPDi_MSkip* **by** *auto*

lemma *LPDi_Plus*: $LPDi\ i\ (MPlus\ r\ s) \subseteq \{MPlus\ r\ s\} \cup LPDi\ i\ r \cup LPDi\ i\ s$
by (*induct i arbitrary: r s*) *auto*

lemma *LPDs_Plus*: $LPDs (MPlus\ r\ s) \subseteq \{MPlus\ r\ s\} \cup LPDs\ r \cup LPDs\ s$
unfolding *LPDs_def* **using** *LPDi_Plus[of _ r s]* **by** *auto*

lemma *LPDi_Times*:
 $LPDi\ i\ (MTimes\ r\ s) \subseteq \{MTimes\ r\ s\} \cup MTimesR\ (\bigcup_{j \leq i} LPDi\ j\ r)\ s \cup (\bigcup_{j \leq i} LPDi\ j\ s)$
proof (*induct i arbitrary: r s*)
case (*Suc i*)
show *?case*
by (*fastforce simp: MTimesR_def dest: bspec[of _ _ Suc _] dest!: set_mp[OF Suc]*)
qed *simp*

lemma *LPDs_Times*: $LPDs (MTimes\ r\ s) \subseteq \{MTimes\ r\ s\} \cup MTimesR\ (LPDs\ r)\ s \cup LPDs\ s$
unfolding *LPDs_def* **using** *LPDi_Times[of _ r s]* **by** (*force simp: MTimesR_def*)

lemma *LPDi_Star*: $j \leq i \implies LPDi\ j\ (MStar\ r) \subseteq \{MStar\ r\} \cup MTimesR\ (\bigcup_{j \leq i} LPDi\ j\ r)\ (MStar\ r)$
proof (*induct i arbitrary: j r*)
case (*Suc i*)
from *Suc(2)* **show** *?case*
by (*auto 0 5 simp: MTimesR_def image_iff le_Suc_eq dest: bspec[of _ _ Suc 0] bspec[of _ _ Suc _] set_mp[OF Suc(1)] dest!: set_mp[OF LPDi_Times]*)
qed *simp*

lemma *LPDs_Star*: $LPDs (MStar\ r) \subseteq \{MStar\ r\} \cup MTimesR\ (LPDs\ r)\ (MStar\ r)$
unfolding *LPDs_def* **using** *LPDi_Star[OF order_refl, of _ r]* **by** (*force simp: MTimesR_def*)

lemma *finite_LPDs*: *finite (LPDs r)*
proof (*induct r*)
case (*MSkip n*)
then show *?case* **by** (*intro finite_subset[OF LPDs_MSkip]*) *simp*
next
case (*MTestPos φ*)
then show *?case* **by** (*intro finite_subset[OF LPDs_Test(2)]*) *simp*
next
case (*MTestNeg φ*)
then show *?case* **by** (*intro finite_subset[OF LPDs_Test(3)]*) *simp*
next
case (*MPlus r s*)
then show *?case* **by** (*intro finite_subset[OF LPDs_Plus]*) *simp*
next
case (*MTimes r s*)
then show *?case* **by** (*intro finite_subset[OF LPDs_Times]*) (*simp add: MTimesR_def*)

```

next
  case (MStar r)
  then show ?case by (intro finite_subset[OF LPDs_Star]) (simp add: MTimesR_def)
qed

context begin

private abbreviation (input) addLPD r ≡ λS. insert r S ∪ Set.bind (insert r S) LPD

private lemma mono_addLPD: mono (addLPD r)
  unfolding mono_def Set.bind_def by auto

private lemma LPDs_aux1: lfp (addLPD r) ⊆ LPDs r
  by (rule lfp_induct[OF mono_addLPD], auto intro: LPDs_refl LPDs_trans simp: Set.bind_def)

private lemma LPDs_aux2: LPDi i r ⊆ lfp (addLPD r)
proof (induct i)
  case 0
  then show ?case
    by (subst lfp_unfold[OF mono_addLPD]) auto
next
  case (Suc i)
  then show ?case
    by (subst lfp_unfold[OF mono_addLPD]) (auto simp: LPDi_Suc_alt simp del: LPDi_simps)
qed

lemma LPDs_alt: LPDs r = lfp (addLPD r)
  using LPDs_aux1 LPDs_aux2 by (force simp: LPDs_def)

lemma LPDs_code[code]:
  LPDs r = saturate (addLPD r) {}
  unfolding LPDs_alt saturate_def
  by (rule lfp_while[OF mono_addLPD _ finite_LPDs, of r]) (auto simp: LPDs_refl LPDs_trans
Set.bind_def)

```

end

6.2.2 RPD

primrec RPD where

```

  RPD (MSkip n) = (case n of 0 ⇒ {} | Suc m ⇒ {MSkip m})
| RPD (MTestPos φ) = {}
| RPD (MTestNeg φ) = {}
| RPD (MPlus r s) = (RPD r ∪ RPD s)
| RPD (MTimes r s) = MTimesL r (RPD s) ∪ RPD r
| RPD (MStar r) = MTimesL (MStar r) (RPD r)

```

primrec RPDi where

```

  RPDi 0 r = {r}
| RPDi (Suc i) r = (⋃ s ∈ RPD r. RPDi i s)

```

lemma RPDi_Suc_alt: $RPDi (Suc i) r = (\bigcup s \in RPD i r. RPD s)$
 by (induct i arbitrary: r) fastforce+

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

lemma RPDs_refl: $r \in RPDs r$

by (auto simp: RPDs_def intro: exI[of _ 0])

lemma RPDs_trans: $r \in RPD s \implies s \in RPDs t \implies r \in RPDs t$

by (force simp: RPDs_def RPDi_Suc_alt simp del: RPDi.simps intro: exI[of _ Suc _])

lemma RPDi_Test:

RPDi i (MSkip 0) \subseteq {MSkip 0}
 RPDi i (MTestPos φ) \subseteq {MTestPos φ }
 RPDi i (MTestNeg φ) \subseteq {MTestNeg φ }
by (induct i) auto

lemma RPDs_Test:

RPDs (MSkip 0) \subseteq {MSkip 0}
 RPDs (MTestPos φ) \subseteq {MTestPos φ }
 RPDs (MTestNeg φ) \subseteq {MTestNeg φ }
unfolding RPDs_def **using** RPDi_Test **by** blast+

lemma RPDi_MSkip: RPDi i (MSkip n) \subseteq MSkip ‘ { i . $i \leq n$ }
by (induct i arbitrary: n) (auto dest: set_mp[OF RPDi_Test(1)] simp: le_Suc_eq split: nat.splits)

lemma RPDs_MSkip: RPDs (MSkip n) \subseteq MSkip ‘ { i . $i \leq n$ }
unfolding RPDs_def **using** RPDi_MSkip **by** auto

lemma RPDi_Plus: RPDi i (MPlus r s) \subseteq {MPlus r s } \cup RPDi i r \cup RPDi i s
by (induct i arbitrary: r s) auto

lemma RPDi_Suc_RPD_Plus:

RPDi (Suc i) r \subseteq RPDs (MPlus r s)
 RPDi (Suc i) s \subseteq RPDs (MPlus r s)
unfolding RPDs_def **by** (force intro!: exI[of _ Suc i])+

lemma RPDs_Plus: RPDs (MPlus r s) \subseteq {MPlus r s } \cup RPDs r \cup RPDs s
unfolding RPDs_def **using** RPDi_Plus[of _ r s] **by** auto

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 (induct i arbitrary: r s)

case (Suc i)

show ?case

by (fastforce simp: MTimesL_def dest: bspec[of _ _ Suc _] dest!: set_mp[OF Suc])

qed simp

lemma RPDs_Times: RPDs (MTimes r s) \subseteq {MTimes r s } \cup MTimesL r (RPDs s) \cup RPDs r
unfolding RPDs_def **using** RPDi_Times[of _ r s] **by** (force simp: MTimesL_def)

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 (induct i arbitrary: j r)

case (Suc i)

from Suc(2) **show** ?case

by (auto 0 5 simp: MTimesL_def image_iff le_Suc_eq

dest: bspec[of _ _ Suc 0] bspec[of _ _ Suc _] set_mp[OF Suc(1)] dest!: set_mp[OF RPDi_Times])

qed simp

lemma RPDs_Star: RPDs (MStar r) \subseteq {MStar r } \cup MTimesL (MStar r) (RPDs r)
unfolding RPDs_def **using** RPDi_Star[OF order_refl, of _ r] **by** (force simp: MTimesL_def)

lemma finite_RPDs: finite (RPDs r)

proof (induct r)

case MSkip

then show ?case **by** (intro finite_subset[OF RPDs_MSkip]) simp

```

next
  case (MTestPos  $\varphi$ )
  then show ?case by (intro finite_subset[OF RPDs_Test(2)]) simp
next
  case (MTestNeg  $\varphi$ )
  then show ?case by (intro finite_subset[OF RPDs_Test(3)]) simp
next
  case (MPlus  $r s$ )
  then show ?case by (intro finite_subset[OF RPDs_Plus]) simp
next
  case (MTimes  $r s$ )
  then show ?case by (intro finite_subset[OF RPDs_Times]) (simp add: MTimesL_def)
next
  case (MStar  $r$ )
  then show ?case by (intro finite_subset[OF RPDs_Star]) (simp add: MTimesL_def)
qed

context begin

private abbreviation (input) addRPD  $r \equiv \lambda S. \text{insert } r S \cup \text{Set.bind } (\text{insert } r S) \text{ RPD}$ 

private lemma mono_addRPD: mono (addRPD  $r$ )
  unfolding mono_def Set.bind_def by auto

private lemma RPDs_aux1: lfp (addRPD  $r$ )  $\subseteq$  RPDs  $r$ 
  by (rule lfp_induct[OF mono_addRPD], auto intro: RPDs_refl RPDs_trans simp: Set.bind_def)

private lemma RPDs_aux2: RPDi  $i r \subseteq$  lfp (addRPD  $r$ )
proof (induct  $i$ )
  case 0
  then show ?case
  by (subst lfp_unfold[OF mono_addRPD]) auto
next
  case (Suc  $i$ )
  then show ?case
  by (subst lfp_unfold[OF mono_addRPD]) (auto simp: RPDi_Suc_alt simp del: RPDi.simps)
qed

lemma RPDs_alt: RPDs  $r =$  lfp (addRPD  $r$ )
  using RPDs_aux1 RPDs_aux2 by (force simp: RPDs_def)

lemma RPDs_code[code]:
  RPDs  $r = \text{saturate } (\text{addRPD } r) \{\}$ 
  unfolding RPDs_alt saturate_def
  by (rule lfp_while[OF mono_addRPD _ finite_RPDs, of  $r$ ]) (auto simp: RPDs_refl RPDs_trans
Set.bind_def)

end

```

6.3 The executable monitor

```

type_synonym  $ts = \text{nat}$ 

type_synonym 'a mbuf2 = 'a table list  $\times$  'a table list
type_synonym 'a mbufn = 'a table list list
type_synonym 'a msaux = ( $ts \times$  'a table) list
type_synonym 'a muaux = ( $ts \times$  'a table  $\times$  'a table) list
type_synonym 'a mr $\delta$ aux = ( $ts \times$  (mregex, 'a table) mapping) list

```

type_synonym 'a ml δ aux = (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
 mr δ aux
 | MMatchF \mathcal{I} mregex mregex list ('msaux, 'muaux) mformula list event_data mbufn ts list event_data
 ml δ aux

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

fun eq_rel :: nat \Rightarrow Formula.trm \Rightarrow Formula.trm \Rightarrow 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: $x \in \text{regex.atms } r \implies \text{size } x < \text{regex.size_regex size } r$
by (induct r) auto

lemma atms_size:
assumes $x \in \text{atms } r$
shows $\text{size } x < \text{Regex.size_regex size } r$

proof –
 { **fix** y **assume** $y \in \text{regex.atms } r$ **case** y of formula.Neg z \Rightarrow x = z | _ \Rightarrow False
then have $\text{size } x < \text{Regex.size_regex size } r$
by (cases y rule: formula.exhaust) (auto dest: regex_atms_size)
 }
with assms **show** ?thesis
unfolding atms_def
by (auto split: formula.splits dest: regex_atms_size)
qed

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* = ($\text{args_ivl} = I, \text{args_n} = n, \text{args_L} = L, \text{args_R} = R, \text{args_pos} = \text{pos}$)

locale *msaux* =

fixes *valid_msaux* :: $\text{args} \Rightarrow \text{ts} \Rightarrow \text{'msaux} \Rightarrow \text{event_data msaux} \Rightarrow \text{bool}$
and *init_msaux* :: $\text{args} \Rightarrow \text{'msaux}$
and *add_new_ts_msaux* :: $\text{args} \Rightarrow \text{ts} \Rightarrow \text{'msaux} \Rightarrow \text{'msaux}$
and *join_msaux* :: $\text{args} \Rightarrow \text{event_data table} \Rightarrow \text{'msaux} \Rightarrow \text{'msaux}$
and *add_new_table_msaux* :: $\text{args} \Rightarrow \text{event_data table} \Rightarrow \text{'msaux} \Rightarrow \text{'msaux}$
and *result_msaux* :: $\text{args} \Rightarrow \text{'msaux} \Rightarrow \text{event_data table}$
assumes *valid_init_msaux*: $L \subseteq R \Longrightarrow$
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* $\Longrightarrow nt \geq cur \Longrightarrow$
valid_msaux *args nt* (*add_new_ts_msaux* *args nt aux*)
(*filter* ($\lambda(t, \text{rel}). \text{enat } (nt - t) \leq \text{right } (\text{args_ivl } \text{args})$) *auxlist*)
assumes *valid_join_msaux*: *valid_msaux* *args cur aux auxlist* \Longrightarrow

fun *check_before* :: $\mathcal{I} \Rightarrow \text{ts} \Rightarrow (\text{ts} \times \text{'a} \times \text{'b}) \Rightarrow \text{bool}$ **where**
check_before *I dt (t, a, b)* $\longleftrightarrow \text{enat } t + \text{right } I < \text{enat } dt$

fun *proj_thd* :: $(\text{'a} \times \text{'b} \times \text{'c}) \Rightarrow \text{'c}$ **where**
proj_thd (*t, a1, a2*) = *a2*

definition *update_until* :: $\text{args} \Rightarrow \text{event_data table} \Rightarrow \text{event_data table} \Rightarrow \text{ts} \Rightarrow \text{event_data muaux} \Rightarrow$
event_data muaux **where**

update_until *args rel1 rel2 nt aux* =
(*map* ($\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } (\text{args_pos } \text{args}) \text{ then } \text{join } a1 \text{ True } \text{rel1} \text{ else } a1 \cup \text{rel1},$
if mem ($nt - t$) (*args_ivl* *args*) *then* $a2 \cup \text{join } \text{rel2 } (\text{args_pos } \text{args}) \text{ } a1 \text{ else } a2$) *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 (*induction auxlist*)

case *Nil*

then show ?*case* **by** (*simp add: update_until_def*)

next

case (*Cons a auxlist*)

then show ?*case*

by (*cases right (args_ivl args)*) (*auto simp add: update_until_def split: if_splits prod.splits*)

qed

fun *eval_until* :: $\mathcal{I} \Rightarrow \text{ts} \Rightarrow \text{event_data muaux} \Rightarrow \text{event_data table list} \times \text{event_data muaux}$ **where**

eval_until *I nt* [] = ([], [])

| *eval_until* *I nt* ((*t, a1, a2*) # *aux*) = (*if* $t + \text{right } I < nt$ *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'
  by (induction I nt auxlist arbitrary: res auxlist' rule: eval_until.induct)
  (auto split: if_splits prod.splits)

lemma eval_until_res: eval_until I nt auxlist = (res, auxlist') ⇒
  res = map proj_thd (takeWhile (check_before I nt) auxlist)
  by (induction I nt auxlist arbitrary: res auxlist' rule: eval_until.induct)
  (auto split: prod.splits)

lemma eval_until_auxlist': eval_until I nt auxlist = (res, auxlist') ⇒
  auxlist' = drop (length res) auxlist
  by (induction I nt auxlist arbitrary: res auxlist' rule: eval_until.induct)
  (auto split: if_splits prod.splits)

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 φ ψ) = MOr (minit0 n φ) (minit0 n ψ) ([], [])
| minit0 n (Formula.And φ ψ) = (if safe_assignment (fv φ) ψ then
  MAndAssign (minit0 n φ) (split_assignment (fv φ) ψ)
  else if safe_formula ψ then
  MAnd (fv φ) (minit0 n φ) True (fv ψ) (minit0 n ψ) ([], [])
  else if is_constraint ψ then
  MAndRel (minit0 n φ) (split_constraint ψ)
  else (case ψ of Formula.Neg ψ ⇒
  MAnd (fv φ) (minit0 n φ) False (fv ψ) (minit0 n ψ) ([], [])))
| 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 φ) = MExists (minit0 (Suc n) φ)
| minit0 n (Formula.Agg y ω b f φ) = MAgg (fv φ ⊆ {0..<b}) y ω b f (minit0 (b + n) φ)
| minit0 n (Formula.Prev I φ) = MPrev I (minit0 n φ) True [] []
| minit0 n (Formula.Next I φ) = MNext I (minit0 n φ) True []
| minit0 n (Formula.Since φ I ψ) = (if safe_formula φ
  then MSince (init_args I n (Formula.fv φ) (Formula.fv ψ) True) (minit0 n φ) (minit0 n ψ) ([], []) []
  (init_msaux (init_args I n (Formula.fv φ) (Formula.fv ψ) True))
  else (case φ of
  Formula.Neg φ ⇒ MSince (init_args I n (Formula.fv φ) (Formula.fv ψ) False) (minit0 n φ) (minit0
  n ψ) ([], []) [] (init_msaux (init_args I n (Formula.fv φ) (Formula.fv ψ) False))
  | _ ⇒ undefined))
| minit0 n (Formula.Until φ I ψ) = (if safe_formula φ
  then MUntil (init_args I n (Formula.fv φ) (Formula.fv ψ) True) (minit0 n φ) (minit0 n ψ) ([], []) []
  (init_muaux (init_args I n (Formula.fv φ) (Formula.fv ψ) True))
  else (case φ of
  Formula.Neg φ ⇒ MUntil (init_args I n (Formula.fv φ) (Formula.fv ψ) False) (minit0 n φ) (minit0
  n ψ) ([], []) [] (init_muaux (init_args I n (Formula.fv φ) (Formula.fv ψ) False))
  | _ ⇒ undefined))
| minit0 n (Formula.MatchP I r) =
  (let (mr, φs) = to_mregex r
  in MMatchP I mr (sorted_list_of_set (RPDs mr)) (map (minit0 n) φs) (replicate (length φs) []) [] [])
| minit0 n (Formula.MatchF I r) =
  (let (mr, φs) = to_mregex r
  in MMatchF I mr (sorted_list_of_set (LPDs mr)) (map (minit0 n) φs) (replicate (length φs) []) [] [])
| minit0 n _ = undefined
by pat_completeness auto
termination (in maux)
by (relation_measure (λ(_, φ). size φ))
  (auto simp: less_Suc_eq_le size_list_estimation' size_remove_neg)

```

dest!: *to_mregex_ok*[*OF sym*] *atms_size*)

definition (in *maux*) *minit* :: *Formula.formula* \Rightarrow ('*msaux*', '*muaux*') *mstate* **where**
minit φ = (let *n* = *Formula.nfv* φ in (\downarrow *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 (*induct xs*)

case (*Cons x xs*)

then show ?*case*

by (*cases xs*) *auto*

qed *simp*

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*))

by *pat_completeness auto*

termination by (*relation measure* (λ (_, _, *buf*). *size_list length buf*))
(*auto simp: comp_def Suc_le_eq size_list_length_diff1*)

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 => None
  | Some f => (case f x of
    None => Some (f(x ↦ y))
    | Some z => if y = z then Some f else None))
| match _ _ = None

```

```

fun meval_trm :: Formula.trm => event_data tuple => 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 => bool => nat => Formula.agg_op => nat => Formula.trm =>
  event_data table => 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 mau_x) 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

```

| $\varepsilon_strict\ n\ \varphi\ s\ (MTimes\ r\ s) = \varepsilon_lax\ (\varepsilon_strict\ n\ \varphi\ s)\ \varphi\ s\ r$
| $\varepsilon_strict\ n\ \varphi\ s\ (MStar\ r) = unit_table\ n$

fun $r\delta :: (mregex \Rightarrow mregex) \Rightarrow (mregex, 'a\ table)\ mapping \Rightarrow 'a\ table\ list \Rightarrow mregex \Rightarrow 'a\ table$ **where**
 $r\delta\ \kappa\ X\ \varphi\ s\ (MSkip\ n) = (case\ n\ of\ 0 \Rightarrow empty_table\ |\ Suc\ m \Rightarrow lookup\ X\ (\kappa\ (MSkip\ m)))$
| $r\delta\ \kappa\ X\ \varphi\ s\ (MTestPos\ i) = empty_table$
| $r\delta\ \kappa\ X\ \varphi\ s\ (MTestNeg\ i) = empty_table$
| $r\delta\ \kappa\ X\ \varphi\ s\ (MPlus\ r\ s) = r\delta\ \kappa\ X\ \varphi\ s\ r\ \cup\ r\delta\ \kappa\ X\ \varphi\ s\ s$
| $r\delta\ \kappa\ X\ \varphi\ s\ (MTimes\ r\ s) = r\delta\ (\lambda t. \kappa\ (MTimes\ r\ t))\ X\ \varphi\ s\ \cup\ \varepsilon_lax\ (r\delta\ \kappa\ X\ \varphi\ s\ r)\ \varphi\ s\ r$
| $r\delta\ \kappa\ X\ \varphi\ s\ (MStar\ r) = r\delta\ (\lambda t. \kappa\ (MTimes\ (MStar\ r)\ t))\ X\ \varphi\ s\ r$

fun $l\delta :: (mregex \Rightarrow mregex) \Rightarrow (mregex, 'a\ table)\ mapping \Rightarrow 'a\ table\ list \Rightarrow mregex \Rightarrow 'a\ table$ **where**
 $l\delta\ \kappa\ X\ \varphi\ s\ (MSkip\ n) = (case\ n\ of\ 0 \Rightarrow empty_table\ |\ Suc\ m \Rightarrow lookup\ X\ (\kappa\ (MSkip\ m)))$
| $l\delta\ \kappa\ X\ \varphi\ s\ (MTestPos\ i) = empty_table$
| $l\delta\ \kappa\ X\ \varphi\ s\ (MTestNeg\ i) = empty_table$
| $l\delta\ \kappa\ X\ \varphi\ s\ (MPlus\ r\ s) = l\delta\ \kappa\ X\ \varphi\ s\ r\ \cup\ l\delta\ \kappa\ X\ \varphi\ s\ s$
| $l\delta\ \kappa\ X\ \varphi\ s\ (MTimes\ r\ s) = l\delta\ (\lambda t. \kappa\ (MTimes\ r\ t))\ X\ \varphi\ s\ \cup\ \varepsilon_lax\ (l\delta\ \kappa\ X\ \varphi\ s\ r)\ \varphi\ s\ r$
| $l\delta\ \kappa\ X\ \varphi\ s\ (MStar\ r) = l\delta\ (\lambda t. \kappa\ (MTimes\ t\ (MStar\ r)))\ X\ \varphi\ s\ r$

lift_definition $mrtabulate :: mregex\ list \Rightarrow (mregex \Rightarrow 'b\ table) \Rightarrow (mregex, 'b\ table)\ mapping$
is $\lambda ks\ f. (map_of\ (List.map_filter\ (\lambda k. let\ fk = f\ k\ in\ if\ fk = empty_table\ then\ None\ else\ Some\ (k,\ fk)))\ ks))$.

lemma $lookup_tabulate$:

$distinct\ xs \Longrightarrow lookup\ (mrtabulate\ xs\ f)\ x = (if\ x \in set\ xs\ then\ f\ x\ else\ empty_table)$

unfolding $lookup_default_def\ lookup_def$

by $transfer\ (auto\ simp:\ Let_def\ map_filter_def\ map_of_eq_None_iff_o_def\ image_image\ dest!\: map_of_SomeD\ split:\ if_splits\ option.splits)$

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

$event_data\ m\delta aux \Rightarrow event_data\ table \times event_data\ m\delta aux$ **where**

$update_matchP\ n\ I\ mr\ mrs\ rels\ nt\ aux =$

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

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

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

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

$| x \# aux' \Rightarrow (if\ fst\ x = nt\ then\ x \# aux'$

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

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

definition $update_matchF_base$ **where**

$update_matchF_base\ n\ I\ mr\ mrs\ rels\ nt =$

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

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

definition $update_matchF_step$ **where**

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

$(let\ Y = mrtabulate\ mrs\ (l\delta\ id\ X\ rels')$

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

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

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

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

| *eval_matchF* *I mr nt* ((*t*, *rels*, *rel*) # *aux*) = (if *t* + *right I* < *nt* then
 (let (*xs*, *aux*) = *eval_matchF I mr nt aux* in (*rel* # *xs*, *aux*)) else ([], (*t*, *rels*, *rel*) # *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* :: *nat* × *Formula.trm* ⇒ *event_data tuple* ⇒ *event_data tuple* **where**
eval_assignment (*x*, *t*) *y* = (*y*[*x*:=*Some* (*meval_trm t y*)])

fun *eval_constraint0* :: *mconstraint* ⇒ *event_data* ⇒ *event_data* ⇒ *bool* **where**

eval_constraint0 *MEq* *x y* = (*x* = *y*)
 | *eval_constraint0* *MLess* *x y* = (*x* < *y*)
 | *eval_constraint0* *MLessEq* *x y* = (*x* ≤ *y*)

fun *eval_constraint* :: *Formula.trm* × *bool* × *mconstraint* × *Formula.trm* ⇒ *event_data tuple* ⇒ *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* :: *nat* ⇒ *ts* ⇒ *Formula.database* ⇒ ('*msaux*, '*muaux*) *mformula* ⇒
event_data table list × ('*msaux*, '*muaux*) *mformula* **where**

meval *n t db* (*MRel* *rel*) = ([*rel*], *MRel* *rel*)
 | *meval* *n t db* (*MPred* *e ts*) = (*map* (λ*X*. (λ*f*. *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* (λ*r1 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* (λ*r*. *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* (λ*xs 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* (λ*r1 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* (λ*r*. (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* (λ*r*. *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  $\varphi$  first nts) =
  (let (xs,  $\varphi$ ) = meval n t db  $\varphi$ ;
    (xs, first) = (case (xs, first) of (_ # xs, True)  $\Rightarrow$  (xs, False) | a  $\Rightarrow$  a);
    (zs, _, nts) = mprev_next I xs (nts @ [t])
  in (zs, MNext I  $\varphi$  first nts))
| meval n t db (MSince args  $\varphi$   $\psi$  buf nts aux) =
  (let (xs,  $\varphi$ ) = meval n t db  $\varphi$ ; (ys,  $\psi$ ) = meval n t db  $\psi$ ;
    ((zs, aux), buf, nts) = mbuf2t_take ( $\lambda r1 r2 t$  (zs, aux).
      let (z, aux) = update_since args r1 r2 t aux
        in (zs @ [z], aux) (mbuf2_add xs ys buf) (nts @ [t]))
  in (zs, MSince args  $\varphi$   $\psi$  buf nts aux))
| meval n t db (MUntil args  $\varphi$   $\psi$  buf nts aux) =
  (let (xs,  $\varphi$ ) = meval n t db  $\varphi$ ; (ys,  $\psi$ ) = meval n t db  $\psi$ ;
    (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 []  $\Rightarrow$  t | nt # _  $\Rightarrow$  nt) aux
  in (zs, MUntil args  $\varphi$   $\psi$  buf nts aux))
| meval n t db (MMatchP I mr mrs  $\varphi$ s buf nts aux) =
  (let (xss,  $\varphi$ s) = map_split id (map (meval n t db)  $\varphi$ s);
    ((zs, aux), buf, nts) = mbufnt_take ( $\lambda$ rels t (zs, aux).
      let (z, aux) = update_matchP n I mr mrs rels t aux
        in (zs @ [z], aux) (mbufn_add xss buf) (nts @ [t]))
  in (zs, MMatchP I mr mrs  $\varphi$ s buf nts aux))
| meval n t db (MMatchF I mr mrs  $\varphi$ s 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 maux) mstep :: Formula.database \times ts \Rightarrow ('msaux, 'muaux) mstate \Rightarrow (nat \times event_data table) list \times ('msaux, 'muaux) mstate **where**

```

mstep tdb st =
  (let (xs, m) = meval (mstate_n st) (snd tdb) (fst tdb) (mstate_m st)
    in (List.enumerate (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 =

```

$\text{Inf } \{i. \forall k. k < j \wedge k \leq \text{min_regex_default } (\text{progress } P) \ r \ j \longrightarrow \tau \ \sigma \ i + \text{right } I \geq \tau \ \sigma \ k\}$

definition $\text{progress_regex } P = \text{min_regex_default } (\text{progress } P)$

declare $\text{progress.simps}[simp \ \text{del}]$

lemmas $\text{progress_simps}[simp] = \text{progress.simps}[\text{folded } \text{progress_regex_def}[\text{THEN } \text{fun_cong}, \text{THEN } \text{fun_cong}]]$

end

definition $\text{pred_mapping } Q = \text{pred_fun } (\lambda_. \text{True}) (\text{pred_option } Q)$

definition $\text{rel_mapping } Q = \text{rel_fun } (=) (\text{rel_option } Q)$

lemma $\text{pred_mapping_alt}: \text{pred_mapping } Q \ P = (\forall p \in \text{dom } P. Q (\text{the } (P \ p)))$
unfolding $\text{pred_mapping_def } \text{pred_fun_def } \text{option.pred_set } \text{dom_def}$
by $(\text{force } \text{split}: \text{option.splits})$

lemma $\text{rel_mapping_alt}: \text{rel_mapping } Q \ P \ P' = (\text{dom } P = \text{dom } P' \wedge (\forall p \in \text{dom } P. Q (\text{the } (P \ p))) (\text{the } (P' \ p))))$
unfolding $\text{rel_mapping_def } \text{rel_fun_def } \text{rel_option_iff } \text{dom_def}$
by $(\text{force } \text{split}: \text{option.splits})$

lemma $\text{rel_mapping_map_upd}[simp]: Q \ x \ y \Longrightarrow \text{rel_mapping } Q \ P \ P' \Longrightarrow \text{rel_mapping } Q \ (P(p \mapsto x)) \ (P'(p \mapsto y))$
by $(\text{auto } \text{simp}: \text{rel_mapping_alt})$

lemma $\text{pred_mapping_map_upd}[simp]: Q \ x \Longrightarrow \text{pred_mapping } Q \ P \Longrightarrow \text{pred_mapping } Q \ (P(p \mapsto x))$
by $(\text{auto } \text{simp}: \text{pred_mapping_alt})$

lemma $\text{pred_mapping_empty}[simp]: \text{pred_mapping } Q \ \text{Map.empty}$
by $(\text{auto } \text{simp}: \text{pred_mapping_alt})$

lemma $\text{pred_mapping_mono}: \text{pred_mapping } Q \ P \Longrightarrow Q \leq R \Longrightarrow \text{pred_mapping } R \ P$
by $(\text{auto } \text{simp}: \text{pred_mapping_alt})$

lemma $\text{pred_mapping_mono_strong}: \text{pred_mapping } Q \ P \Longrightarrow (\bigwedge p. p \in \text{dom } P \Longrightarrow Q (\text{the } (P \ p)) \Longrightarrow R (\text{the } (P \ p))) \Longrightarrow \text{pred_mapping } R \ P$
by $(\text{auto } \text{simp}: \text{pred_mapping_alt})$

lemma $\text{progress_mono_gen}: j \leq j' \Longrightarrow \text{rel_mapping } (\leq) \ P \ P' \Longrightarrow \text{progress } \sigma \ P \ \varphi \ j \leq \text{progress } \sigma \ P' \ \varphi \ j'$

proof $(\text{induction } \varphi \ \text{arbitrary}: P \ P')$
case $(\text{Pred } e \ ts)$
then show $?case$
by $(\text{force } \text{simp}: \text{rel_mapping_alt } \text{dom_def } \text{split}: \text{option.splits})$

next
case $(\text{Ands } l)$
then show $?case$
by $(\text{auto } \text{simp}: \text{image_iff } \text{intro!}: \text{Min.coboundedI}[\text{THEN } \text{order_trans}])$

next
case $(\text{Until } \varphi \ I \ \psi)$
from $\text{Until}(1,2)[\text{of } P \ P'] \ \text{Until.premis } \text{show } ?case$
by $(\text{cases } \text{right } I)$
 $(\text{auto } \text{dest}: \text{trans_le_add1}[\text{OF } \tau_mono] \ \text{intro!}: \text{cInf_superset_mono})$

next
case $(\text{MatchF } I \ r)$
from $\text{MatchF}(1)[\text{of } _ \ P \ P'] \ \text{MatchF.premis } \text{show } ?case$
by $(\text{cases } \text{right } I; \ \text{cases } \text{regex.atms } r = \{\})$
 $(\text{auto } 0 \ 3 \ \text{simp}: \text{Min_le_iff } \text{progress_regex_def } \text{dest}: \text{trans_le_add1}[\text{OF } \tau_mono])$

```

    intro!: cInf_superset_mono elim!: less_le_trans order_trans)
qed (force simp: Min_le_iff progress_regex_def split: option.splits)+

lemma rel_mapping_refl: reflp Q  $\implies$  rel_mapping Q P P
  by (auto simp: rel_mapping_alt reflp_def)

lemmas progress_mono = progress_mono_gen[OF _ rel_mapping_refl[unfolded reflp_def], simplified]

lemma progress_le_gen: pred_mapping ( $\lambda x. x \leq j$ ) P  $\implies$  progress  $\sigma$  P  $\varphi$  j  $\leq$  j
proof (induction  $\varphi$  arbitrary: P)
  case (Pred e ts)
  then show ?case
    by (auto simp: pred_mapping_alt dom_def split: option.splits)
next
  case (Ands l)
  then show ?case
    by (auto simp: image_iff intro!: Min.coboundedI[where a=progress  $\sigma$  P (hd l) j, THEN order_trans])
next
  case (Until  $\varphi$  I  $\psi$ )
  then show ?case
    by (cases right I)
      (auto intro: trans_le_add1[OF  $\tau$ _mono] intro!: cInf_lower)
next
  case (MatchF I r)
  then show ?case
    by (cases right I)
      (auto intro: trans_le_add1[OF  $\tau$ _mono] intro!: cInf_lower)
qed (force simp: Min_le_iff progress_regex_def split: option.splits)+

lemma progress_le: progress  $\sigma$  Map.empty  $\varphi$  j  $\leq$  j
  using progress_le_gen[of _ Map.empty] by auto

lemma progress_0_gen[simp]:
  pred_mapping ( $\lambda x. x = 0$ ) P  $\implies$  progress  $\sigma$  P  $\varphi$  0 = 0
  using progress_le_gen[of 0 P] by auto

lemma progress_0[simp]:
  progress  $\sigma$  Map.empty  $\varphi$  0 = 0
  by (auto simp: pred_mapping_alt)

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

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

lemma dom_max_mapping[simp]: dom (max_mapping P1 P2) = dom P1  $\cap$  dom P2
  unfolding max_mapping_def by (auto split: option.splits)

lemma dom_Max_mapping[simp]: dom (Max_mapping X) = ( $\bigcap P \in X. dom P$ )
  unfolding Max_mapping_def by (auto split: if_splits)

lemma Max_mapping_coboundedI:
  assumes finite X  $\forall Q \in X. dom Q = dom P$  P  $\in$  X

```

shows $rel_mapping (\leq) P (Max_mapping X)$
unfolding $rel_mapping_alt$
proof (*intro conjI ballI*)
from $assms(3)$ **have** $X \neq \{\}$ **by** *auto*
then show $dom P = dom (Max_mapping X)$ **using** $assms(2)$ **by** *auto*
next
fix p
assume $p \in dom P$
with $assms$ **show** $the (P p) \leq the (Max_mapping X p)$
by (*force simp add: Max_mapping_def intro!: Max.coboundedI imageI*)
qed

lemma $rel_mapping_trans: P OO Q \leq R \implies$
 $rel_mapping P P1 P2 \implies rel_mapping Q P2 P3 \implies rel_mapping R P1 P3$
by (*force simp: rel_mapping_alt dom_def set_eq_iff*)

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 $range_mapping_relax:$
 $range_mapping i j P \implies i' \leq i \implies j' \geq j \implies range_mapping i' j' P$
by (*auto simp: pred_mapping_alt dom_def set_eq_iff max_mapping_def split: option.splits*)

lemma $range_mapping_max_mapping[simp]:$
 $range_mapping i j1 P1 \implies range_mapping i j2 P2 \implies range_mapping i (max j1 j2) (max_mapping P1 P2)$
by (*auto simp: pred_mapping_alt dom_def set_eq_iff max_mapping_def split: option.splits*)

lemma $range_mapping_Max_mapping[simp]:$
 $finite X \implies X \neq \{\} \implies \forall x \in X. range_mapping i (j x) (P x) \implies range_mapping i (Max (j ` X)) (Max_mapping (P ` X))$
by (*force simp: pred_mapping_alt Max_mapping_def dom_def image_iff intro!: Max_ge_iff[THEN iffD2] split: if_splits*)

lemma $pred_mapping_le:$
 $pred_mapping ((\leq) i) P1 \implies rel_mapping (\leq) P1 P2 \implies pred_mapping ((\leq) (i :: nat)) P2$
by (*force simp: rel_mapping_alt pred_mapping_alt dom_def set_eq_iff*)

lemma $pred_mapping_le':$
 $pred_mapping ((\leq) j) P1 \implies i \leq j \implies rel_mapping (\leq) P1 P2 \implies pred_mapping ((\leq) (i :: nat)) P2$
by (*force simp: rel_mapping_alt pred_mapping_alt dom_def set_eq_iff*)

lemma $max_mapping_cobounded1: dom P1 \subseteq dom P2 \implies rel_mapping (\leq) P1 (max_mapping P1 P2)$
unfolding $max_mapping_def$ $rel_mapping_alt$ **by** (*auto simp: dom_def split: option.splits*)

lemma $max_mapping_cobounded2: dom P2 \subseteq dom P1 \implies rel_mapping (\leq) P2 (max_mapping P1 P2)$
unfolding $max_mapping_def$ $rel_mapping_alt$ **by** (*auto simp: dom_def split: option.splits*)

lemma $max_mapping_fun_upd2[simp]:$
 $(max_mapping P1 (P2(p := y)))(p \mapsto x) = (max_mapping P1 P2)(p \mapsto x)$
by (*auto simp: max_mapping_def*)

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

```

by (auto simp: rel_mapping_alt max_mapping_def split: option.splits)

lemma progress_ge_gen: Formula.future_bounded  $\varphi \implies$ 
   $\exists P j. \text{dom } P = S \wedge \text{range\_mapping } i j P \wedge i \leq \text{progress } \sigma P \varphi j$ 
proof (induction  $\varphi$  arbitrary:  $i S$ )
  case (Pred e ts)
  then show ?case
  by (intro exI[of _  $\lambda e. \text{if } e \in S \text{ then Some } i \text{ else None}$ ])
    (auto split: option.splits if_splits simp: rel_mapping_alt pred_mapping_alt dom_def)
next
  case (Let p  $\varphi \psi$ )
  from Let.prem1 obtain  $P2 j2$  where  $P2: \text{dom } P2 = \text{insert } p S \text{ range\_mapping } i j2 P2$ 
     $i \leq \text{progress } \sigma P2 \psi j2$ 
  by (atomize_elim, intro Let(2)) (force simp: pred_mapping_alt rel_mapping_alt dom_def)+
  from Let.prem2 obtain  $P1 j1$  where  $P1: \text{dom } P1 = S \text{ range\_mapping } (the (P2 p)) j1 P1$ 
     $the (P2 p) \leq \text{progress } \sigma P1 \varphi j1$ 
  by (atomize_elim, intro Let(1)) auto
  let ?P12 = max_mapping P1 P2
  from P1 P2 have le1:  $\text{progress } \sigma P1 \varphi j1 \leq \text{progress } \sigma (?P12(p := P1 p)) \varphi (max j1 j2)$ 
  by (intro progress_mono_gen) (auto simp: rel_mapping_alt max_mapping_def)
  from P1 P2 have le2:  $\text{progress } \sigma P2 \psi j2 \leq \text{progress } \sigma (?P12(p \mapsto \text{progress } \sigma P1 \varphi j1)) \psi (max j1 j2)$ 
  by (intro progress_mono_gen) (auto simp: rel_mapping_alt max_mapping_def)
  show ?case
  unfolding progress_simps
  proof (intro exI[of _ ?P12(p := P1 p)] exI[of _  $max j1 j2$ ] conjI)
    show  $\text{dom } (?P12(p := P1 p)) = S$ 
    using P1 P2 by (auto simp: dom_def max_mapping_def)
  next
    show  $\text{range\_mapping } i (max j1 j2) (?P12(p := P1 p))$ 
    using P1 P2 by (force simp add: pred_mapping_alt dom_def max_mapping_def split: option.splits)
  next
    have  $i \leq \text{progress } \sigma P2 \psi j2$  by fact
    also have  $\dots \leq \text{progress } \sigma (?P12(p \mapsto \text{progress } \sigma P1 \varphi j1)) \psi (max j1 j2)$ 
    using le2 by blast
    also have  $\dots \leq \text{progress } \sigma ((?P12(p := P1 p))(p \mapsto \text{progress } \sigma (?P12(p := P1 p)) \varphi (max j1 j2))) \psi (max j1 j2)$ 
    by (auto intro!: progress_mono_gen simp: le1 rel_mapping_alt)
    finally show  $i \leq \dots$  .
  qed
next
  case (Eq _ _)
  then show ?case
  by (intro exI[of _  $\lambda e. \text{if } e \in S \text{ then Some } i \text{ else None}$ ]) (auto split: if_splits simp: pred_mapping_alt)
next
  case (Less _ _)
  then show ?case
  by (intro exI[of _  $\lambda e. \text{if } e \in S \text{ then Some } i \text{ else None}$ ]) (auto split: if_splits simp: pred_mapping_alt)
next
  case (LessEq _ _)
  then show ?case
  by (intro exI[of _  $\lambda e. \text{if } e \in S \text{ then Some } i \text{ else None}$ ]) (auto split: if_splits simp: pred_mapping_alt)
next
  case (Or  $\varphi1 \varphi2$ )
  from Or(3) obtain  $P1 j1$  where  $P1: \text{dom } P1 = S \text{ range\_mapping } i j1 P1$   $i \leq \text{progress } \sigma P1 \varphi1 j1$ 
  using Or(1)[of  $S i$ ] by auto
  moreover
  from Or(3) obtain  $P2 j2$  where  $P2: \text{dom } P2 = S \text{ range\_mapping } i j2 P2$   $i \leq \text{progress } \sigma P2 \varphi2 j2$ 

```

```

    using Or(2)[of S i] by auto
ultimately have  $i \leq \text{progress } \sigma (\text{max\_mapping } P1 P2) (\text{Formula.Or } \varphi1 \varphi2) (\text{max } j1 j2)$ 
  by (auto 0 3 elim!: order.trans[OF _ progress_mono_gen] intro: max_mapping_cobounded1 max_mapping_cobounded2)
with P1 P2 show ?case by (intro exI[of _ max_mapping P1 P2] exI[of _ max j1 j2]) auto
next
case (And  $\varphi1 \varphi2$ )
from And(3) obtain P1 j1 where P1:  $\text{dom } P1 = S \text{ range\_mapping } i j1 P1 \ i \leq \text{progress } \sigma P1 \ \varphi1 \ j1$ 
  using And(1)[of S i] by auto
moreover
from And(3) obtain P2 j2 where P2:  $\text{dom } P2 = S \text{ range\_mapping } i j2 P2 \ i \leq \text{progress } \sigma P2 \ \varphi2 \ j2$ 
  using And(2)[of S i] by auto
ultimately have  $i \leq \text{progress } \sigma (\text{max\_mapping } P1 P2) (\text{Formula.And } \varphi1 \varphi2) (\text{max } j1 j2)$ 
  by (auto 0 3 elim!: order.trans[OF _ progress_mono_gen] intro: max_mapping_cobounded1 max_mapping_cobounded2)
with P1 P2 show ?case by (intro exI[of _ max_mapping P1 P2] exI[of _ max j1 j2]) auto
next
case (Ands l)
show ?case proof (cases l = [])
  case True
  then show ?thesis
    by (intro exI[of _  $\lambda e. \text{if } e \in S \text{ then Some } i \text{ else None}$ ]
      (auto split: if_splits simp: pred_mapping_alt))
  next
  case False
  then obtain  $\varphi$  where  $\varphi \in \text{set } l$  by (cases l) auto
  from Ands.prem1 have  $\forall \varphi \in \text{set } l. \text{Formula.future\_bounded } \varphi$ 
    by (simp add: list.pred_set)
  { fix  $\varphi$ 
    assume  $\varphi \in \text{set } l$ 
    with Ands.prem1 obtain P j where  $\text{dom } P = S \text{ range\_mapping } i j P \ i \leq \text{progress } \sigma P \ \varphi \ j$ 
      by (atomize_elim, intro Ands(1)[of  $\varphi S i$ ]) (auto simp: list.pred_set)
    then have  $\exists Pj. \text{dom } (\text{fst } Pj) = S \wedge \text{range\_mapping } i (\text{snd } Pj) (\text{fst } Pj) \wedge i \leq \text{progress } \sigma (\text{fst } Pj) \ \varphi$ 
      (snd Pj)
      (is  $\exists Pj. ?P Pj$ )
      by (intro exI[of _ (P, j)]) auto
    }
  then have  $\forall \varphi \in \text{set } l. \exists Pj. \text{dom } (\text{fst } Pj) = S \wedge \text{range\_mapping } i (\text{snd } Pj) (\text{fst } Pj) \wedge i \leq \text{progress } \sigma$ 
    (fst Pj)  $\varphi$  (snd Pj)
    (is  $\forall \varphi \in \text{set } l. \exists Pj. ?P Pj \ \varphi$ )
    by blast
  with Ands(1) Ands.prem1 False have  $\exists Pjf. \forall \varphi \in \text{set } l. ?P (Pjf \ \varphi) \ \varphi$ 
    by (auto simp: Ball_def intro: choice)
  then obtain Pjf where  $Pjf: \forall \varphi \in \text{set } l. ?P (Pjf \ \varphi) \ \varphi \dots$ 
  define Pf where  $Pf = \text{fst } o \ Pjf$ 
  define jf where  $jf = \text{snd } o \ Pjf$ 
  have *:  $\text{dom } (Pf \ \varphi) = S \text{ range\_mapping } i (jf \ \varphi) (Pf \ \varphi) \ i \leq \text{progress } \sigma (Pf \ \varphi) \ \varphi (jf \ \varphi)$ 
    if  $\varphi \in \text{set } l$  for  $\varphi$ 
    using Pjf[THEN bspec, OF that] unfolding Pf_def jf_def by auto
  with False show ?thesis
    unfolding progress_simps eq_False[THEN iffD2, OF False] if_False
    by ((subst Min_ge_iff; simp add: False),
      intro exI[where  $x = \text{MAX } \varphi \in \text{set } l. jf \ \varphi$ ] exI[where  $x = \text{Max\_mapping } (Pf \ \varphi \ \text{set } l)$ ]
      conjI ballI order.trans[OF *(3) progress_mono_gen] Max_mapping_coboundedI]
      (auto simp: False *[OF  $\langle \varphi \in \text{set } l \rangle \langle \varphi \in \text{set } l \rangle$ ])
qed
next
case (Exists  $\varphi$ )
then show ?case by simp
next

```

```

case (Prev I  $\varphi$ )
then obtain P j where dom P = S range_mapping i j P i  $\leq$  progress  $\sigma$  P  $\varphi$  j
  by (atomize_elim, intro Prev(1)) (auto simp: pred_mapping_alt dom_def)
with Prev(2) have
  dom P = S  $\wedge$  range_mapping i (max i j) P  $\wedge$  i  $\leq$  progress  $\sigma$  P (formula.Prev I  $\varphi$ ) (max i j)
  by (auto simp: le_Suc_eq max_def pred_mapping_alt split: if_splits
    elim: order.trans[OF progress_mono])
then show ?case by blast
next
case (Next I  $\varphi$ )
then obtain P j where dom P = S range_mapping (Suc i) j P Suc i  $\leq$  progress  $\sigma$  P  $\varphi$  j
  by (atomize_elim, intro Next(1)) (auto simp: pred_mapping_alt dom_def)
then show ?case
  by (intro exI[of _ P] exI[of _ j]) (auto 0 3 simp: pred_mapping_alt dom_def)
next
case (Since  $\varphi$ 1 I  $\varphi$ 2)
from Since(3) obtain P1 j1 where P1: dom P1 = S range_mapping i j1 P1 i  $\leq$  progress  $\sigma$  P1  $\varphi$ 1 j1
  using Since(1)[of S i] by auto
moreover
from Since(3) obtain P2 j2 where P2: dom P2 = S range_mapping i j2 P2 i  $\leq$  progress  $\sigma$  P2  $\varphi$ 2 j2
  using Since(2)[of S i] by auto
ultimately have i  $\leq$  progress  $\sigma$  (max_mapping P1 P2) (Formula.Since  $\varphi$ 1 I  $\varphi$ 2) (max j1 j2)
  by (auto elim!: order.trans[OF progress_mono_gen] simp: max_mapping_cobounded1 max_mapping_cobounded2)
with P1 P2 show ?case by (intro exI[of _ max_mapping P1 P2] exI[of _ max j1 j2])
  (auto elim!: pred_mapping_le intro: max_mapping_cobounded1)
next
case (Until  $\varphi$ 1 I  $\varphi$ 2)
from Until.premis obtain b where [simp]: right I = enat b
  by (cases right I) (auto)
obtain i' where i < i' and  $\tau \sigma i + b + 1 \leq \tau \sigma i'$ 
  using ex_le_ $\tau$ [where x= $\tau \sigma i + b + 1$ ] by (auto simp add: less_eq_Suc_le)
then have 1:  $\tau \sigma i + b < \tau \sigma i'$  by simp
from Until.premis obtain P1 j1 where P1: dom P1 = S range_mapping (Suc i') j1 P1 Suc i'  $\leq$ 
progress  $\sigma$  P1  $\varphi$ 1 j1
  by (atomize_elim, intro Until(1)) (auto simp: pred_mapping_alt dom_def)
from Until.premis obtain P2 j2 where P2: dom P2 = S range_mapping (Suc i') j2 P2 Suc i'  $\leq$ 
progress  $\sigma$  P2  $\varphi$ 2 j2
  by (atomize_elim, intro Until(2)) (auto simp: pred_mapping_alt dom_def)
let ?P12 = max_mapping P1 P2
have i  $\leq$  progress  $\sigma$  ?P12 (Formula.Until  $\varphi$ 1 I  $\varphi$ 2) (max j1 j2)
  unfolding progress_simps
proof (intro cInf_greatest, goal_cases nonempty greatest)
  case nonempty
  then show ?case
    by (auto simp: trans_le_add1[OF  $\tau$ _mono] intro!: exI[of _ max j1 j2])
next
case (greatest x)
from P1(2,3) have i' < j1
  by (auto simp: less_eq_Suc_le intro!: progress_le_gen elim!: order.trans pred_mapping_mono_strong)
then have i' < max j1 j2 by simp
have progress  $\sigma$  P1  $\varphi$ 1 j1  $\leq$  progress  $\sigma$  ?P12  $\varphi$ 1 (max j1 j2)
  using P1(1) P2(1) by (auto intro!: progress_mono_gen max_mapping_cobounded1)
moreover have progress  $\sigma$  P2  $\varphi$ 2 j2  $\leq$  progress  $\sigma$  ?P12  $\varphi$ 2 (max j1 j2)
  using P1(1) P2(1) by (auto intro!: progress_mono_gen max_mapping_cobounded2)
ultimately have i'  $\leq$  min (progress  $\sigma$  ?P12  $\varphi$ 1 (max j1 j2)) (progress  $\sigma$  ?P12  $\varphi$ 2 (max j1 j2))
  using P1(3) P2(3) by simp
with greatest (i' < max j1 j2) have  $\tau \sigma i' \leq \tau \sigma x + b$ 
  by simp

```

```

with 1 have  $\tau \sigma i < \tau \sigma x$  by simp
then show ?case by (auto dest!: less_τD)
qed
with P1 P2 ⟨i < i'⟩ show ?case
  by (intro exI[of _ max_mapping P1 P2] exI[of _ max j1 j2]) (auto simp: range_mapping_relax)
next
case (MatchP I r)
then show ?case
proof (cases regex.atms r = {})
  case True
  with MatchP.premis show ?thesis
    unfolding progress.simps
    by (intro exI[of _ λe. if e ∈ S then Some i else None] exI[of _ i])
      (auto split: if_splits simp: pred_mapping_alt regex.pred_set)
  next
  case False
  define pick where pick = (λφ. SOME Pj. dom (fst Pj) = S ∧ range_mapping i (snd Pj) (fst Pj) ∧
    i ≤ progress σ (fst Pj) φ (snd Pj))
  let ?pickP = fst o pick let ?pickj = snd o pick
  from MatchP have pick: φ ∈ regex.atms r ⇒ dom (?pickP φ) = S ∧
    range_mapping i (?pickj φ) (?pickP φ) ∧ i ≤ progress σ (?pickP φ) φ (?pickj φ) for φ
  unfolding pick_def o_def future_bounded.simps regex.pred_set
  by (intro someI_ex[where P = λPj. dom (fst Pj) = S ∧ range_mapping i (snd Pj) (fst Pj) ∧
    i ≤ progress σ (fst Pj) φ (snd Pj)]) auto
  with False show ?thesis
    unfolding progress.simps
    by (intro exI[of _ Max_mapping (?pickP ' regex.atms r)] exI[of _ Max (?pickj ' regex.atms r)])
      (auto simp: Max_mapping_coboundedI
        order_trans[OF pick[THEN conjunct2, THEN conjunct2] progress_mono_gen])
  qed
next
case (MatchF I r)
from MatchF.premis obtain b where [simp]: right I = enat b
  by auto
obtain i' where i': i < i' τ σ i + b + 1 ≤ τ σ i'
  using ex_le_τ[where x=τ σ i + b + 1] by (auto simp add: less_eq_Suc_le)
then have 1: τ σ i + b < τ σ i' by simp
have ix: i ≤ x if τ σ i' ≤ b + τ σ x b + τ σ i < τ σ i' for x
  using less_τD[of σ i] that less_le_trans by fastforce
show ?case
proof (cases regex.atms r = {})
  case True
  with MatchF.premis i' ix show ?thesis
    unfolding progress.simps
    by (intro exI[of _ λe. if e ∈ S then Some (Suc i') else None] exI[of _ Suc i'])
      (auto split: if_splits simp: pred_mapping_alt regex.pred_set add.commute less_Suc_eq
        intro!: cInf_greatest dest!: spec[of _ i'] less_imp_le[THEN τ_mono, of _ i' σ])
  next
  case False
  then obtain φ where φ: φ ∈ regex.atms r by auto
  define pick where pick = (λφ. SOME Pj. dom (fst Pj) = S ∧ range_mapping (Suc i') (snd Pj) (fst
Pj) ∧
    Suc i' ≤ progress σ (fst Pj) φ (snd Pj))
  define pickP where pickP = fst o pick define pickj where pickj = snd o pick
  from MatchF have pick: φ ∈ regex.atms r ⇒ dom (pickP φ) = S ∧
    range_mapping (Suc i') (pickj φ) (pickP φ) ∧ Suc i' ≤ progress σ (pickP φ) φ (pickj φ) for φ
  unfolding pick_def o_def future_bounded.simps regex.pred_set pickj_def pickP_def
  by (intro someI_ex[where P = λPj. dom (fst Pj) = S ∧ range_mapping (Suc i') (snd Pj) (fst Pj)

```

```

 $\wedge$ 
  Suc i'  $\leq$  progress  $\sigma$  (fst Pj)  $\varphi$  (snd Pj)) auto
let ?P = Max_mapping (pickP ' regex.atms r) let ?j = Max (pickj ' regex.atms r)
from pick[OF  $\varphi$ ] False  $\varphi$  have Suc i'  $\leq$  ?j
by (intro order_trans[OF pick[THEN conjunct2, THEN conjunct2], OF  $\varphi$ ] order_trans[OF progress_le_gen])
  (auto simp: Max_ge_iff dest: range_mapping_relax[of ___ 0, OF ___ order_refl, simplified])
moreover
note i' 1 ix
moreover
from MatchF.premis have Regex.pred_regex Formula.future_bounded r
  by auto
ultimately show ?thesis using  $\tau$ _mono[of ___ ?j  $\sigma$ ] less_ $\tau$ D[of  $\sigma$  i] pick False
  by (intro exI[of ___ ?j] exI[of ___ ?P])
  (auto 0 3 intro!: cInf_greatest
    order_trans[OF le_SucI[OF order_refl] order_trans[OF pick[THEN conjunct2, THEN conjunct2]]
progress_mono_gen]]
  range_mapping_Max_mapping[OF ___ ball[OF range_mapping_relax[of Suc i' ___ i, OF ___
  ___ order_refl]]]
  simp: ac_simps Suc_le_eq trans_le_add2 Max_mapping_coboundedI progress_regex_def
  dest: spec[of ___ i'] spec[of ___ ?j])
qed
qed (auto split: option.splits)

lemma progress_ge: Formula.future_bounded  $\varphi$   $\implies$   $\exists j. i \leq$  progress  $\sigma$  Map.empty  $\varphi$  j
  using progress_ge_gen[of  $\varphi$  {} i  $\sigma$ ]
  by auto

lemma cInf_restrict_nat:
  fixes x :: nat
  assumes x  $\in$  A
  shows Inf A = Inf {y  $\in$  A. y  $\leq$  x}
  using assms by (auto intro!: antisym intro: cInf_greatest cInf_lower Inf_nat_def1)

lemma progress_time_conv:
  assumes  $\forall i < j. \tau \sigma i = \tau \sigma' i$ 
  shows progress  $\sigma$  P  $\varphi$  j = progress  $\sigma'$  P  $\varphi$  j
  using assms proof (induction  $\varphi$  arbitrary: P)
  case (Until  $\varphi$ 1 I  $\varphi$ 2)
  have *: i  $\leq$  j - 1  $\iff$  i < j if j  $\neq$  0 for i
    using that by auto
  with Until show ?case
  proof (cases right I)
  case (enat b)
  then show ?thesis
  proof (cases j)
  case (Suc n)
  with enat * Until show ?thesis
    using  $\tau$ _mono[THEN trans_le_add1]
    by (auto 8 0
      intro!: box_equals[OF arg_cong[where f=Inf]
cInf_restrict_nat[symmetric, where x=n] cInf_restrict_nat[symmetric, where x=n]])
  qed simp
  qed simp
next
case (MatchF I r)
have *: i  $\leq$  j - 1  $\iff$  i < j if j  $\neq$  0 for i
  using that by auto
with MatchF show ?case using  $\tau$ _mono[THEN trans_le_add1]

```

```

    by (cases right I; cases j)
      ((auto 6 0 simp: progress_le_gen progress_regeq_def intro!: box_equals[OF arg_cong[where f=Inf]
        cInf_restrict_nat[symmetric, where x=j-1] cInf_restrict_nat[symmetric, where x=j-1]])
    [])+
  qed (auto simp: progress_regeq_def)

lemma Inf_UNIV_nat: (Inf UNIV :: nat) = 0
  by (simp add: cInf_eq_minimum)

lemma progress_prefix_conv:
  assumes prefix_of  $\pi$   $\sigma$  and prefix_of  $\pi$   $\sigma'$ 
  shows progress  $\sigma$   $P$   $\varphi$  (plen  $\pi$ ) = progress  $\sigma'$   $P$   $\varphi$  (plen  $\pi$ )
  using assms by (auto intro: progress_time_conv  $\tau$ _prefix_conv)

lemma bounded_rtranclp_mono:
  fixes  $n :: 'x :: linorder$ 
  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 rtranclp  $R$   $i$   $j \implies j < n \implies rtranclp S i j$ 
  proof (induct rule: rtranclp_induct)
    case (step  $y z$ )
    then show ?case
      using assms(1,2)[of  $y z$ ]
      by (auto elim!: rtrancl_into_rtrancl[to_pred, rotated])
  qed auto

lemma sat_prefix_conv_gen:
  assumes prefix_of  $\pi$   $\sigma$  and prefix_of  $\pi$   $\sigma'$ 
  shows  $i < \text{progress } \sigma P \varphi (\text{plen } \pi) \implies \text{dom } V = \text{dom } V' \implies \text{dom } P = \text{dom } V \implies$ 
    pred_mapping  $(\lambda x. x \leq \text{plen } \pi) P \implies$ 
     $(\bigwedge p i \varphi. p \in \text{dom } V \implies i < \text{the } (P p) \implies \text{the } (V p) i = \text{the } (V' p) i) \implies$ 
    Formula.sat  $\sigma V v i \varphi \iff \text{Formula.sat } \sigma' V' v i \varphi$ 
  proof (induction  $\varphi$  arbitrary:  $P V V' v i$ )
    case (Pred  $e$   $ts$ )
    from Pred.prem(1,4) have  $i < \text{plen } \pi$ 
      by (blast intro!: order.strict_trans2 progress_le_gen)
    show ?case proof (cases  $V e$ )
      case None
      then have  $V' e = \text{None}$  using  $\langle \text{dom } V = \text{dom } V' \rangle$  by auto
      with None  $\Gamma$ _prefix_conv[OF assms(1,2)  $\langle i < \text{plen } \pi \rangle$ ] show ?thesis by simp
    next
      case (Some  $a$ )
      obtain  $a'$  where  $V' e = \text{Some } a'$  using  $\langle \text{dom } V = \text{dom } V' \rangle$  by auto
      then have  $i < \text{the } (P e)$ 
        using Pred.prem(1-3) by (auto split: option.splits)
      then have  $\text{the } (V e) i = \text{the } (V' e) i$ 
        using Some by (intro Pred.prem(5)) (simp_all add: domI)
      with Some  $\langle V' e = \text{Some } a' \rangle$  show ?thesis by simp
    qed
  qed
next
  case (Let  $p \varphi \psi$ )
  let ?V =  $\lambda V \sigma. (V(p \mapsto \lambda i. \{v. \text{length } v = \text{Formula.nfv } \varphi \wedge \text{Formula.sat } \sigma V v i \varphi\}))$ 
  show ?case unfolding sat.simps proof (rule Let.IH(2))
    from Let.prem show  $i < \text{progress } \sigma (P(p \mapsto \text{progress } \sigma P \varphi (\text{plen } \pi))) \psi (\text{plen } \pi)$ 
      by simp
    from Let.prem show  $\text{dom } (?V V \sigma) = \text{dom } (?V V' \sigma')$ 
      by simp
    from Let.prem show  $\text{dom } (P(p \mapsto \text{progress } \sigma P \varphi (\text{plen } \pi))) = \text{dom } (?V V \sigma)$ 
      by simp
  end

```

```

from Let.prems show pred_mapping ( $\lambda x. x \leq \text{plen } \pi$ ) ( $P(p \mapsto \text{progress } \sigma P \varphi (\text{plen } \pi))$ )
  by (auto intro!: pred_mapping_map_upd elim!: progress_le_gen)
next
  fix  $p' i \varphi'$ 
  assume  $1: p' \in \text{dom} (?V V \sigma)$  and  $2: i < \text{the} ((P(p \mapsto \text{progress } \sigma P \varphi (\text{plen } \pi))) p')$ 
  show  $\text{the} (?V V \sigma p') i = \text{the} (?V V' \sigma' p') i$  proof (cases  $p' = p$ )
    case True
      with Let 2 show ?thesis by auto
    next
      case False
        with 1 2 show ?thesis by (auto intro!: Let.prems(5))
    qed
  qed
next
  case (Eq t1 t2)
  show ?case by simp
next
  case (Neg  $\varphi$ )
  then show ?case by simp
next
  case (Or  $\varphi 1 \varphi 2$ )
  then show ?case by auto
next
  case (Ands  $l$ )
  from Ands.prems have  $\forall \varphi \in \text{set } l. i < \text{progress } \sigma P \varphi (\text{plen } \pi)$ 
  by (cases  $l$ ) simp_all
  with Ands show ?case unfolding sat_Ands by blast
next
  case (Exists  $\varphi$ )
  then show ?case by simp
next
  case (Prev  $I \varphi$ )
  with  $\tau\_prefix\_conv[OF \text{ assms}(1,2)]$  show ?case
  by (cases  $i$ ) (auto split: if_splits)
next
  case (Next  $I \varphi$ )
  then have  $Suc\ i < \text{plen } \pi$ 
  by (auto intro: order.strict_trans2[OF _ progress_le_gen[of _ P \sigma \varphi]])
  with Next.prems  $\tau\_prefix\_conv[OF \text{ assms}(1,2)]$  show ?case
  unfolding sat.simps
  by (intro conj_cong Next) auto
next
  case (Since  $\varphi 1 I \varphi 2$ )
  then have  $i < \text{plen } \pi$ 
  by (auto elim!: order.strict_trans2[OF _ progress_le_gen])
  with Since.prems  $\tau\_prefix\_conv[OF \text{ assms}(1,2)]$  show ?case
  unfolding sat.simps
  by (intro conj_cong ex_cong ball_cong Since) auto
next
  case (Until  $\varphi 1 I \varphi 2$ )
  from Until.prems obtain  $b$  where right[simp]: right I = enat b
  by (cases right I) (auto simp add: Inf_UNIV_nat)
  from Until.prems obtain  $j$  where  $\tau \sigma i + b + 1 \leq \tau \sigma j$ 
   $j \leq \text{progress } \sigma P \varphi 1 (\text{plen } \pi)$   $j \leq \text{progress } \sigma P \varphi 2 (\text{plen } \pi)$ 
  by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le intro: Suc_leI dest: spec[of _ i])
  dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have  $1: k < \text{progress } \sigma P \varphi 1 (\text{plen } \pi)$  and  $2: k < \text{progress } \sigma P \varphi 2 (\text{plen } \pi)$ 
  if  $\tau \sigma k \leq \tau \sigma i + b$  for  $k$ 

```

```

using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of σ])+
have 3:  $k < \text{plen } \pi$  if  $\tau \sigma k \leq \tau \sigma i + b$  for  $k$ 
using 1[OF that] Until(6) by (auto simp only: less_eq_Suc_le order.trans[OF _ progress_le_gen])

from Until.prems have  $i < \text{progress } \sigma' P$  (Formula.Until  $\varphi 1 I \varphi 2$ ) ( $\text{plen } \pi$ )
unfolding progress_prefix_conv[OF assms(1,2)] by blast
then obtain  $j$  where  $\tau \sigma' i + b + 1 \leq \tau \sigma' j$ 
 $j \leq \text{progress } \sigma' P \varphi 1$  ( $\text{plen } \pi$ )  $j \leq \text{progress } \sigma' P \varphi 2$  ( $\text{plen } \pi$ )
by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le intro: Suc_leI dest: spec[of _ i]
dest!: le_cInf_iff[THEN iffD1, rotated -1])
then have 11:  $k < \text{progress } \sigma P \varphi 1$  ( $\text{plen } \pi$ ) and 21:  $k < \text{progress } \sigma P \varphi 2$  ( $\text{plen } \pi$ )
if  $\tau \sigma' k \leq \tau \sigma' i + b$  for  $k$ 
unfolding progress_prefix_conv[OF assms(1,2)]
using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of σ'])+
have 31:  $k < \text{plen } \pi$  if  $\tau \sigma' k \leq \tau \sigma' i + b$  for  $k$ 
using 11[OF that] Until(6) by (auto simp only: less_eq_Suc_le order.trans[OF _ progress_le_gen])
show ?case unfolding sat.simps
proof ((intro ex_cong iffI; elim conjE), goal_cases LR RL)
case (LR j)
with Until(1)[OF 1] Until(2)[OF 2] τ_prefix_conv[OF assms(1,2) 3] Until.prems show ?case
by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF τ_mono, rotated])
next
case (RL j)
with Until(1)[OF 11] Until(2)[OF 21] τ_prefix_conv[OF assms(1,2) 31] Until.prems show ?case
by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF τ_mono, rotated])
qed
next
case (MatchP I r)
then have  $i < \text{plen } \pi$ 
by (force simp: pred_mapping_alt elim!: order.strict_trans2[OF _ progress_le_gen])
with MatchP.prems τ_prefix_conv[OF assms(1,2)] show ?case
unfolding sat.simps
by (intro ex_cong conj_cong match_cong_strong MatchP) (auto simp: progress_regex_def split:
if_splits)
next
case (MatchF I r)
from MatchF.prems obtain  $b$  where right[simp]:  $\text{right } I = \text{enat } b$ 
by (cases right I) (auto simp add: Inf_UNIV_nat)
show ?case
proof (cases regex.atms r = {})
case True
from MatchF.prems(1) obtain  $j$  where  $\tau \sigma i + b + 1 \leq \tau \sigma j$   $j \leq \text{plen } \pi$ 
by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le dest!: le_cInf_iff[THEN iffD1, rotated
-1])
then have 1:  $k < \text{plen } \pi$  if  $\tau \sigma k \leq \tau \sigma i + b$  for  $k$ 
using that le_less_trans [of  $\langle \tau \sigma k \rangle$  _  $\langle \tau \sigma j \rangle$ ] less_τD [of  $\sigma k j$ ] by simp
from MatchF.prems have  $i < \text{progress } \sigma' P$  (Formula.MatchF I r) ( $\text{plen } \pi$ )
unfolding progress_prefix_conv[OF assms(1,2)] by blast
then obtain  $j$  where  $\tau \sigma' i + b + 1 \leq \tau \sigma' j$   $j \leq \text{plen } \pi$ 
by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le dest!: le_cInf_iff[THEN iffD1, rotated
-1])
then have 2:  $k < \text{plen } \pi$  if  $\tau \sigma' k \leq \tau \sigma' i + b$  for  $k$ 
using that le_less_trans [of  $\langle \tau \sigma' k \rangle$  _  $\langle \tau \sigma' j \rangle$ ] less_τD [of  $\sigma' k j$ ] by simp
from MatchF.prems(1,4) True show ?thesis
unfolding sat.simps conj_commute[of left I ≤ _ ≤ _]
proof (intro ex_cong conj_cong match_cong_strong, goal_cases left right sat)
case (left j)
then show ?case

```

```

    by (intro iffI)
      ((subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 1, symmetric]; auto elim: order.trans[OF  $\tau\_mono$ ,
rotated]),
      (subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 2]; auto elim: order.trans[OF  $\tau\_mono$ , rotated]))
  next
  case (right j)
  then show ?case
  by (intro iffI)
    ((subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 2, symmetric]; auto elim: order.trans[OF  $\tau\_mono$ ,
rotated]),
    (subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 2]; auto elim: order.trans[OF  $\tau\_mono$ , rotated]))
  qed auto
next
case False
from MatchF.prem(1) False obtain j where  $\tau \sigma i + b + 1 \leq \tau \sigma j$  ( $\forall x \in regex.atms r. j \leq progress$ 
 $\sigma P x$  (plen  $\pi$ ))
  by atomize_elim (auto 0 6 simp add: less_eq_Suc_le not_le progress_regex_def
    dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have 1:  $\varphi \in regex.atms r \implies k < progress \sigma P \varphi$  (plen  $\pi$ ) if  $\tau \sigma k \leq \tau \sigma i + b$  for  $k \varphi$ 
  using that
  by (fastforce elim!: order.strict_trans2[rotated] intro: less_ $\tau D$ [of  $\sigma$ ])
  then have 2:  $k < plen \pi$  if  $\tau \sigma k \leq \tau \sigma i + b$   $regex.atms r \neq \{\}$  for  $k$ 
  using that
  by (fastforce intro: order.strict_trans2[OF _ progress_le_gen[OF MatchF(5), of  $\sigma$ ], of  $k$ ])

from MatchF.prem have  $i < progress \sigma' P$  (Formula.MatchF I r) (plen  $\pi$ )
  unfolding progress_prefix_conv[OF assms(1,2)] by blast
with False obtain j where  $\tau \sigma' i + b + 1 \leq \tau \sigma' j$  ( $\forall x \in regex.atms r. j \leq progress \sigma' P x$  (plen  $\pi$ ))
  by atomize_elim (auto 0 6 simp add: less_eq_Suc_le not_le progress_regex_def
    dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have 11:  $\varphi \in regex.atms r \implies k < progress \sigma P \varphi$  (plen  $\pi$ ) if  $\tau \sigma' k \leq \tau \sigma' i + b$  for  $k \varphi$ 
  using that using progress_prefix_conv[OF assms(1,2)]
  by (auto 0 3 elim!: order.strict_trans2[rotated] intro: less_ $\tau D$ [of  $\sigma'$ ])
  have 21:  $k < plen \pi$  if  $\tau \sigma' k \leq \tau \sigma' i + b$  for  $k$ 
  using 11[OF that(1)] False by (fastforce intro: order.strict_trans2[OF _ progress_le_gen[OF
MatchF(5), of  $\sigma$ ], of  $k$ ])
  show ?thesis unfolding sat.simps conj_commute[of left  $I \leq \_ \leq \_$ ]
  proof ((intro ex_cong conj_cong match_cong_strong MatchF(1)[OF _ MatchF(3-6)]; assumption?), goal_cases right left progress)
  case (right j)
  with False show ?case
  by (intro iffI)
    ((subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 2, symmetric]; auto elim: order.trans[OF  $\tau\_mono$ ,
rotated]),
    (subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 21]; auto elim: order.trans[OF  $\tau\_mono$ , rotated]))
  next
  case (left j)
  with False show ?case unfolding right_enat_ord_code le_diff_conv add commute[of b]
  by (intro iffI)
    ((subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 21, symmetric]; auto elim: order.trans[OF  $\tau\_mono$ ,
rotated]),
    (subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 21]; auto elim: order.trans[OF  $\tau\_mono$ , rotated]))
  next
  case (progress j k z)
  with False show ?case unfolding right_enat_ord_code le_diff_conv add commute[of b]
  by (elim 1[rotated])
    (subst (1 2)  $\tau\_prefix\_conv$ [OF assms(1,2) 21]; auto elim!: order.trans[OF  $\tau\_mono$ , rotated])
  qed

```

qed
qed auto

lemma sat_prefix_conv:
assumes *prefix_of* π σ **and** *prefix_of* π σ'
shows $i < \text{progress } \sigma \text{ Map.empty } \varphi (\text{plen } \pi) \implies$
 $\text{Formula.sat } \sigma \text{ Map.empty } v \ i \ \varphi \longleftrightarrow \text{Formula.sat } \sigma' \text{ Map.empty } v \ i \ \varphi$
by (*erule sat_prefix_conv_gen*[OF *assms*]) *auto*

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

lemma safe_progress_get_and: $\text{safe_formula } \varphi \implies$
 $\text{Min } ((\lambda\varphi. \text{progress } \sigma \ P \ \varphi \ j) \ ' \ \text{set } (\text{get_and_list } \varphi)) = \text{progress } \sigma \ P \ \varphi \ j$
by (*induction* φ *rule: get_and_list.induct*) *auto*

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 (*induction* φ *arbitrary: P rule: safe_formula_induct*)

case (*And_safe* $\varphi \ \psi$)

let $?c = \text{convert_multiway } (\text{Formula.And } \varphi \ \psi)$

let $?c\varphi = \text{convert_multiway } \varphi$

let $?c\psi = \text{convert_multiway } \psi$

have $c_eq: ?c = \text{Formula.Ands } (\text{get_and_list } ?c\varphi \ @ \ \text{get_and_list } ?c\psi)$

using *And_safe* **by** *simp*

from $\langle \text{safe_formula } \varphi \rangle$ **have** $\text{safe_formula } ?c\varphi$ **by** (*rule safe_convert_multiway*)

moreover from $\langle \text{safe_formula } \psi \rangle$ **have** $\text{safe_formula } ?c\psi$ **by** (*rule safe_convert_multiway*)

ultimately show $?case$

unfolding c_eq

using *And_safe.IH*

by (*auto simp: get_and_nonempty Min.union safe_progress_get_and*)

next

case (*And_Not* $\varphi \ \psi$)

let $?c = \text{convert_multiway } (\text{Formula.And } \varphi \ (\text{Formula.Neg } \psi))$

let $?c\varphi = \text{convert_multiway } \varphi$

let $?c\psi = \text{convert_multiway } \psi$

have $c_eq: ?c = \text{Formula.Ands } (\text{Formula.Neg } ?c\psi \ \# \ \text{get_and_list } ?c\varphi)$

using *And_Not* **by** *simp*

from $\langle \text{safe_formula } \varphi \rangle$ **have** $\text{safe_formula } ?c\varphi$ **by** (*rule safe_convert_multiway*)

moreover from $\langle \text{safe_formula } \psi \rangle$ **have** $\text{safe_formula } ?c\psi$ **by** (*rule safe_convert_multiway*)

ultimately show $?case$

unfolding c_eq

using *And_Not.IH*

by (*auto simp: get_and_nonempty Min.union safe_progress_get_and*)

next

case (*MatchP* $I \ r$)

from *MatchP* **show** $?case$

unfolding *progress.simps regex.map convert_multiway.simps regex.set_map image_image*

by (*intro if_cong arg_cong[of __ Min] image_cong*)

(*auto 0 4 simp: atms_def elim!: disjE_Not2 dest: safe_regex_safe_formula*)

next

case (*MatchF* $I \ r$)

from *MatchF* **show** $?case$

unfolding *progress.simps regex.map convert_multiway.simps regex.set_map image_image*

by (*intro if_cong arg_cong[of __ Min] arg_cong[of __ Inf] arg_cong[of __ (\leq) _]*

image_cong Collect_cong all_cong1 imp_cong conj_cong image_cong)

(*auto 0 4 simp: atms_def elim!: disjE_Not2 dest: safe_regex_safe_formula*)

qed auto

6.5 Specification

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

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

lemma $pprogress_eq: prefix_of \pi \sigma \Longrightarrow pprogress \varphi \pi = progress \sigma Map.empty \varphi (plen \pi)$

unfolding $pprogress_def$ **using** $progress_prefix_conv$
by $blast$

locale $future_bounded_mfodl =$

fixes $\varphi :: Formula.formula$

assumes $future_bounded: Formula.future_bounded \varphi$

sublocale $future_bounded_mfodl \subseteq sliceable_timed_progress Formula.nfv \varphi Formula.fv \varphi relevant_events$

φ

$\lambda \sigma v i. Formula.sat \sigma Map.empty v i \varphi pprogress \varphi$

proof ($unfold_locales, goal_cases$)

case (1 x)

then show $?case$ **by** ($simp add: fvi_less_nfv$)

next

case (2 $v v' \sigma i$)

then show $?case$ **by** ($simp cong: sat_fv_cong[rule_format]$)

next

case (3 $v S \sigma i$)

then show $?case$

using $sat_slice_iff[symmetric]$ **by** $simp$

next

case (4 $\pi \pi'$)

moreover obtain σ **where** $prefix_of \pi' \sigma$

using $ex_prefix_of ..$

moreover have $prefix_of \pi \sigma$

using $prefix_of_antimono[OF \langle \pi \leq \pi' \rangle \langle prefix_of \pi' \sigma \rangle]$.

ultimately show $?case$

by ($simp add: pprogress_eq plen_mono progress_mono$)

next

case (5 σx)

obtain j **where** $x \leq progress \sigma Map.empty \varphi j$

using $future_bounded progress_ge$ **by** $blast$

then have $x \leq pprogress \varphi (take_prefix j \sigma)$

by ($simp add: pprogress_eq[of_ \sigma]$)

then show $?case$ **by** $force$

next

case (6 $\pi \sigma \sigma' i v$)

then have $i < progress \sigma Map.empty \varphi (plen \pi)$

by ($simp add: pprogress_eq$)

with 6 **show** $?case$

using sat_prefix_conv **by** $blast$

next

case (7 $\pi \pi'$)

then have $plen \pi = plen \pi'$

by $transfer (simp add: list_eq_iff_nth_eq)$

moreover obtain $\sigma \sigma'$ **where** $prefix_of \pi \sigma prefix_of \pi' \sigma'$

using ex_prefix_of **by** $blast+$

moreover have $\forall i < plen \pi. \tau \sigma i = \tau \sigma' i$

using 7 $calculation$

by $transfer (simp add: list_eq_iff_nth_eq)$

ultimately show $?case$

by ($simp add: pprogress_eq progress_time_conv$)

qed

```

locale verimon_spec =
  fixes  $\varphi$  :: Formula.formula
  assumes monitorable: mmonitorable  $\varphi$ 

sublocale verimon_spec  $\subseteq$  future_bounded_mfodl
  using monitorable by unfold_locales (simp add: mmonitorable_def)

```

6.6 Correctness

6.6.1 Invariants

definition wf_mbuf2 :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow (nat \Rightarrow event_data\ table \Rightarrow bool) \Rightarrow (nat \Rightarrow event_data\ table \Rightarrow bool) \Rightarrow$

```

  event_data mbuf2  $\Rightarrow$  bool where
  wf_mbuf2 i ja jb P Q buf  $\longleftrightarrow$   $i \leq ja \wedge i \leq jb \wedge$  (case buf of (xs, ys)  $\Rightarrow$ 
    list_all2 P [i.. $ja$ ] xs  $\wedge$  list_all2 Q [i.. $jb$ ] ys)

```

inductive list_all3 :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list \Rightarrow bool$ **for** P :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow bool)$ **where**

```

  list_all3 P [] [] []
| P a1 a2 a3  $\Longrightarrow$  list_all3 P q1 q2 q3  $\Longrightarrow$  list_all3 P (a1 # q1) (a2 # q2) (a3 # q3)

```

lemma list_all3_list_all2D: list_all3 P xs ys zs \Longrightarrow
 (length xs = length ys \wedge list_all2 (case_prod P) (zip xs ys) zs)
by (induct xs ys zs rule: list_all3.induct) auto

lemma list_all2_list_all3I: length xs = length ys \Longrightarrow list_all2 (case_prod P) (zip xs ys) zs \Longrightarrow
 list_all3 P xs ys zs
by (induct xs ys arbitrary: zs rule: list_induct2)
 (auto simp: list_all2_Cons1 intro: list_all3.intros)

lemma list_all3_list_all2_eq: list_all3 P xs ys zs \longleftrightarrow
 (length xs = length ys \wedge list_all2 (case_prod P) (zip xs ys) zs)
using list_all2_list_all3I list_all3_list_all2D **by** blast

lemma list_all3_mapD: list_all3 P (map f xs) (map g ys) (map h zs) \Longrightarrow
 list_all3 $(\lambda x y z. P (f x) (g y) (h z))$ xs ys zs
by (induct map f xs map g ys map h zs arbitrary: xs ys zs rule: list_all3.induct)
 (auto intro: list_all3.intros)

lemma list_all3_mapI: list_all3 $(\lambda x y z. P (f x) (g y) (h z))$ xs ys zs \Longrightarrow
 list_all3 P (map f xs) (map g ys) (map h zs)
by (induct xs ys zs rule: list_all3.induct)
 (auto intro: list_all3.intros)

lemma list_all3_map_iff: list_all3 P (map f xs) (map g ys) (map h zs) \longleftrightarrow
 list_all3 $(\lambda x y z. P (f x) (g y) (h z))$ xs ys zs
using list_all3_mapD list_all3_mapI **by** blast

lemmas list_all3_map =
 list_all3_map_iff[**where** g=id **and** h=id, unfolded list.map_id id_apply]
 list_all3_map_iff[**where** f=id **and** h=id, unfolded list.map_id id_apply]
 list_all3_map_iff[**where** f=id **and** g=id, unfolded list.map_id id_apply]

lemma list_all3_conv_all_nth:
 list_all3 P xs ys zs =
 (length xs = length ys \wedge length ys = length zs \wedge $(\forall i < length\ xs. P (xs!i) (ys!i) (zs!i))$)
by (auto simp add: list_all3_list_all2_eq list_all2_conv_all_nth)

lemma *list_all3_refl* [intro?]:
 $(\bigwedge x. x \in \text{set } xs \implies P x x) \implies \text{list_all3 } P \text{ } xs \text{ } xs$
by (*simp add: list_all3_conv_all_nth*)

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**

$wf_mbufn \ i \ js \ Ps \ buf \longleftrightarrow \text{list_all3 } (\lambda P \ j \ xs. i \leq j \wedge \text{list_all2 } P \ [i..<j] \ xs) \ Ps \ js \ buf$

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**
 $wf_mbuf2' \ \sigma \ P \ V \ j \ n \ R \ \varphi \ \psi \ buf \longleftrightarrow wf_mbuf2 \ (\min (\text{progress } \sigma \ P \ \varphi \ j) (\text{progress } \sigma \ P \ \psi \ j))$
 $(\text{progress } \sigma \ P \ \varphi \ j) (\text{progress } \sigma \ P \ \psi \ j)$
 $(\lambda i. \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi))$
 $(\lambda i. \text{qtable } n \ (\text{Formula.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \psi)) \ buf$

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

lemma *wf_mbuf2'_UNIV_alt*: $wf_mbuf2' \ \sigma \ P \ V \ j \ n \ \text{UNIV } \varphi \ \psi \ buf \longleftrightarrow (\text{case } buf \ \text{of } (xs, ys) \Rightarrow$
 $\text{list_all2 } (\lambda i. \text{wf_table } n \ (\text{Formula.fv } \varphi) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi))$
 $[\min (\text{progress } \sigma \ P \ \varphi \ j) (\text{progress } \sigma \ P \ \psi \ j) ..< (\text{progress } \sigma \ P \ \varphi \ j)] \ xs \wedge$
 $\text{list_all2 } (\lambda i. \text{wf_table } n \ (\text{Formula.fv } \psi) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \psi))$
 $[\min (\text{progress } \sigma \ P \ \varphi \ j) (\text{progress } \sigma \ P \ \psi \ j) ..< (\text{progress } \sigma \ P \ \psi \ j)] \ ys)$
unfolding *wf_mbuf2'_def wf_mbuf2_def*
by (*simp add: mem_restr_UNIV[THEN eqTrueI, abs_def] split: prod.split*)

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**

$wf_ts \ \sigma \ P \ j \ \varphi \ \psi \ ts \longleftrightarrow \text{list_all2 } (\lambda i \ t. t = \tau \sigma \ i) [\min (\text{progress } \sigma \ P \ \varphi \ j) (\text{progress } \sigma \ P \ \psi \ j) ..< j] \ ts$

definition *wf_ts_regex* :: $\text{Formula.trace} \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{Formula.formula} \text{ Regex.regex} \Rightarrow \text{ts list} \Rightarrow \text{bool}$ **where**

$wf_ts_regex \ \sigma \ P \ j \ r \ ts \longleftrightarrow \text{list_all2 } (\lambda i \ t. t = \tau \sigma \ i) [\text{progress_regex } \sigma \ P \ r \ j ..< j] \ ts$

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

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

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

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

lemma qtable_mem_restr_UNIV: qtable n A(mem_restr UNIV) Q X = wf_table n A Q X
unfolding qtable_def by auto

lemma (in msaux) wf_since_aux_UNIV_alt:

wf_since_aux σ V UNIV args φ ψ aux ne \longleftrightarrow Formula.fv φ \subseteq Formula.fv ψ \wedge (\exists cur auxlist.
 valid_msaux args cur aux auxlist \wedge
 cur = (if ne = 0 then 0 else τ σ (ne - 1)) \wedge
 sorted_wrt (λ x y. fst x > fst y) auxlist \wedge
 (\forall t X. (t, X) \in set auxlist \longrightarrow ne \neq 0 \wedge t \leq τ σ (ne - 1) \wedge τ σ (ne - 1) - t \leq right (args_ivl
 args) \wedge (\exists i. τ σ i = t) \wedge
 wf_table (args_n args) (Formula.fv ψ)
 (λ v. Formula.sat σ V (map the v) (ne - 1) (Sincep (args_pos args) φ (point (τ σ (ne - 1) -
 t)) ψ)) X) \wedge
 (\forall t. ne \neq 0 \wedge t \leq τ σ (ne - 1) \wedge τ σ (ne - 1) - t \leq right (args_ivl args) \wedge (\exists i. τ σ i = t) \longrightarrow
 (\exists X. (t, X) \in set auxlist)))

unfolding wf_since_aux_def qtable_mem_restr_UNIV ..

definition wf_until_auxlist :: Formula.trace \Rightarrow _ \Rightarrow nat \Rightarrow event_data list set \Rightarrow bool \Rightarrow

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

definition (in muaux) wf_until_aux :: Formula.trace \Rightarrow _ \Rightarrow event_data list set \Rightarrow args \Rightarrow

Formula.formula \Rightarrow Formula.formula \Rightarrow 'muaux \Rightarrow nat \Rightarrow bool **where**
 wf_until_aux σ V R args φ ψ aux ne \longleftrightarrow Formula.fv φ \subseteq Formula.fv ψ \wedge
 (\exists cur auxlist. valid_muaux args cur aux auxlist \wedge
 cur = (if ne + length auxlist = 0 then 0 else τ σ (ne + length auxlist - 1)) \wedge
 wf_until_auxlist σ V (args_n args) R (args_pos args) φ (args_ivl args) ψ auxlist ne)

lemma (in muaux) wf_until_aux_UNIV_alt:

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

$(\forall k \in \{i..<j\}. \text{if } (\text{args_pos } \text{args}) \text{ then } \text{Formula.sat } \sigma V (\text{map the } v) k \varphi \text{ else } \neg \text{Formula.sat } \sigma V (\text{map the } v) k \varphi)) \text{ r2}$
 $\text{auxlist } [\text{ne}..<\text{ne}+\text{length auxlist}]$
unfolding $\text{wf_until_aux_def wf_until_auxlist_def qtable_mem_restr_UNIV ..}$

definition $\text{wf_matchF_aux} :: \text{Formula.trace} \Rightarrow _ \Rightarrow \text{nat} \Rightarrow \text{event_data list set} \Rightarrow$
 $\mathcal{I} \Rightarrow \text{Formula.formula} \text{Regex.regex} \Rightarrow \text{event_data ml}\delta\text{aux} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $\text{wf_matchF_aux } \sigma V n R I r \text{aux ne k} \longleftrightarrow (\text{case to_mregex } r \text{ of } (mr, \varphi s) \Rightarrow$
 $\text{list_all2 } (\lambda x i. \text{case } x \text{ of } (t, \text{rels}, \text{rel}) \Rightarrow t = \tau \sigma i \wedge$
 $\text{list_all2 } (\lambda \varphi. \text{qtable } n (\text{Formula.fv } \varphi) (\text{mem_restr } R) (\lambda v.$
 $\text{Formula.sat } \sigma V (\text{map the } v) i \varphi)) \varphi s \text{rels} \wedge$
 $\text{qtable } n (\text{Formula.fv_regex } r) (\text{mem_restr } R) (\lambda v. (\exists j. i \leq j \wedge j < \text{ne} + \text{length aux} + k \wedge \text{mem}$
 $(\tau \sigma j - \tau \sigma i) I \wedge$
 $\text{Regex.match } (\text{Formula.sat } \sigma V (\text{map the } v)) r i j)) \text{rel}$
 $\text{aux } [\text{ne}..<\text{ne}+\text{length aux}]$)

definition wf_matchF_invar **where**
 $\text{wf_matchF_invar } \sigma V n R I r st i =$
 $(\text{case } st \text{ of } (\text{aux}, Y) \Rightarrow \text{aux} \neq [] \wedge \text{wf_matchF_aux } \sigma V n R I r \text{aux } i 0 \wedge$
 $(\text{case to_mregex } r \text{ of } (mr, \varphi s) \Rightarrow \forall ms \in \text{LPDs } mr.$
 $\text{qtable } n (\text{Formula.fv_regex } r) (\text{mem_restr } R) (\lambda v.$
 $\text{Regex.match } (\text{Formula.sat } \sigma V (\text{map the } v)) (\text{from_mregex } ms \varphi s) i (i + \text{length aux} - 1)) (\text{lookup}$
 $Y ms)))$

definition lift_envs' **where**
 $\text{lift_envs' } b R = (\lambda(xs,ys). xs @ ys) ' (\{xs. \text{length } xs = b\} \times R)$

fun $\text{formula_of_constraint} :: \text{Formula.trm} \times \text{bool} \times \text{mconstraint} \times \text{Formula.trm} \Rightarrow \text{Formula.formula}$
where
 $\text{formula_of_constraint } (t1, \text{True}, \text{MEq}, t2) = \text{Formula.Eq } t1 t2$
 $|\ \text{formula_of_constraint } (t1, \text{True}, \text{MLess}, t2) = \text{Formula.Less } t1 t2$
 $|\ \text{formula_of_constraint } (t1, \text{True}, \text{MLessEq}, t2) = \text{Formula.LessEq } t1 t2$
 $|\ \text{formula_of_constraint } (t1, \text{False}, \text{MEq}, t2) = \text{Formula.Neg } (\text{Formula.Eq } t1 t2)$
 $|\ \text{formula_of_constraint } (t1, \text{False}, \text{MLess}, t2) = \text{Formula.Neg } (\text{Formula.Less } t1 t2)$
 $|\ \text{formula_of_constraint } (t1, \text{False}, \text{MLessEq}, t2) = \text{Formula.Neg } (\text{Formula.LessEq } t1 t2)$

inductive (in maux) $\text{wf_mformula} :: \text{Formula.trace} \Rightarrow \text{nat} \Rightarrow _ \Rightarrow _ \Rightarrow$
 $\text{nat} \Rightarrow \text{event_data list set} \Rightarrow ('msaux, 'muaux) \text{mformula} \Rightarrow \text{Formula.formula} \Rightarrow \text{bool}$
for σj **where**
 $\text{Eq: is_simple_eq } t1 t2 \Longrightarrow$
 $\forall x \in \text{Formula.fv_trm } t1. x < n \Longrightarrow \forall x \in \text{Formula.fv_trm } t2. x < n \Longrightarrow$
 $\text{wf_mformula } \sigma j P V n R (\text{MRel } (\text{eq_rel } n t1 t2)) (\text{Formula.Eq } t1 t2)$
 $|\ \text{neg_Var: } x < n \Longrightarrow$
 $\text{wf_mformula } \sigma j P V n R (\text{MRel empty_table}) (\text{Formula.Neg } (\text{Formula.Eq } (\text{Formula.Var } x) (\text{Formula.Var } x)))$
 $|\ \text{Pred: } \forall x \in \text{Formula.fv } (\text{Formula.Pred } e \text{ts}). x < n \Longrightarrow$
 $\forall t \in \text{set } \text{ts}. \text{Formula.is_Var } t \vee \text{Formula.is_Const } t \Longrightarrow$
 $\text{wf_mformula } \sigma j P V n R (\text{MPred } e \text{ts}) (\text{Formula.Pred } e \text{ts})$
 $|\ \text{Let: } \text{wf_mformula } \sigma j P V m \text{UNIV } \varphi \varphi' \Longrightarrow$
 $\text{wf_mformula } \sigma j (P(p \mapsto \text{progress } \sigma P \varphi' j))$
 $(V(p \mapsto \lambda i. \{v. \text{length } v = m \wedge \text{Formula.sat } \sigma V v i \varphi'\})) n R \psi \psi' \Longrightarrow$
 $\{0..<m\} \subseteq \text{Formula.fv } \varphi' \Longrightarrow b \leq m \Longrightarrow m = \text{Formula.nfv } \varphi' \Longrightarrow$
 $\text{wf_mformula } \sigma j P V n R (\text{MLet } p m \varphi \psi) (\text{Formula.Let } p \varphi' \psi')$
 $|\ \text{And: } \text{wf_mformula } \sigma j P V n R \varphi \varphi' \Longrightarrow \text{wf_mformula } \sigma j P V n R \psi \psi' \Longrightarrow$
 $\text{if pos then } \chi = \text{Formula.And } \varphi' \psi'$
 $\text{else } \chi = \text{Formula.And } \varphi' (\text{Formula.Neg } \psi') \wedge \text{Formula.fv } \psi' \subseteq \text{Formula.fv } \varphi' \Longrightarrow$
 $\text{wf_mbuf2' } \sigma P V j n R \varphi' \psi' \text{buf} \Longrightarrow$
 $\text{wf_mformula } \sigma j P V n R (\text{MAnd } (\text{fv } \varphi') \varphi \text{pos } (\text{fv } \psi') \psi \text{buf}) \chi$

$| \text{AndAssign: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$
 $x < n \implies x \notin Formula.fv \ \varphi' \implies Formula.fv_trm \ t \subseteq Formula.fv \ \varphi' \implies (x, t) = conf \implies$
 $\psi' = Formula.Eq \ (Formula.Var \ x) \ t \vee \psi' = Formula.Eq \ t \ (Formula.Var \ x) \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MAndAssign \ \varphi \ conf) \ (Formula.And \ \varphi' \ \psi')$

$| \text{AndRel: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$
 $\psi' = formula_of_constraint \ conf \implies$
 $(let \ (t1, _, _, \ t2) = conf \ in \ Formula.fv_trm \ t1 \cup \ Formula.fv_trm \ t2 \subseteq \ Formula.fv \ \varphi') \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MAndRel \ \varphi \ conf) \ (Formula.And \ \varphi' \ \psi')$

$| \text{Ands: } list_all2 \ (\lambda\varphi \ \varphi'. \ wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi') \ l \ (l_pos \ @ \ map \ remove_neg \ l_neg) \implies$
 $wf_mbufn \ (progress \ \sigma \ P \ \psi \ j) \ (map \ (\lambda\psi. \ progress \ \sigma \ P \ \psi \ j) \ (l_pos \ @ \ map \ remove_neg \ l_neg)) \ (map \ (\lambda\psi \ i.$
 $\quad qtable \ n \ (Formula.fv \ \psi) \ (mem_restr \ R) \ (\lambda v. \ Formula.sat \ \sigma \ V \ (map \ the \ v) \ i \ \psi)) \ (l_pos \ @ \ map$
 $\quad remove_neg \ l_neg)) \ buf \implies$
 $(l_pos, \ l_neg) = partition \ safe_formula \ l' \implies$
 $l_pos \neq [] \implies$
 $list_all \ safe_formula \ (map \ remove_neg \ l_neg) \implies$
 $A_pos = map \ fv \ l_pos \implies$
 $A_neg = map \ fv \ l_neg \implies$
 $\bigcup \ (set \ A_neg) \subseteq \bigcup \ (set \ A_pos) \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MAnds \ A_pos \ A_neg \ l \ buf) \ (Formula.Ands \ l')$

$| \text{Or: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \psi \ \psi' \implies$
 $Formula.fv \ \varphi' = Formula.fv \ \psi' \implies$
 $wf_mbuf2' \ \sigma \ P \ V \ j \ n \ R \ \varphi' \ \psi' \ buf \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MOr \ \varphi \ \psi \ buf) \ (Formula.Or \ \varphi' \ \psi')$

$| \text{Neg: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies Formula.fv \ \varphi' = \{\} \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MNeg \ \varphi) \ (Formula.Neg \ \varphi')$

$| \text{Exists: } wf_mformula \ \sigma \ j \ P \ V \ (Suc \ n) \ (lift_envs \ R) \ \varphi \ \varphi' \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MExists \ \varphi) \ (Formula.Exists \ \varphi')$

$| \text{Agg: } wf_mformula \ \sigma \ j \ P \ V \ (b + n) \ (lift_envs' \ b \ R) \ \varphi \ \varphi' \implies$
 $y < n \implies$
 $y + b \notin Formula.fv \ \varphi' \implies$
 $\{0..<b\} \subseteq Formula.fv \ \varphi' \implies$
 $Formula.fv_trm \ f \subseteq Formula.fv \ \varphi' \implies$
 $g0 = (Formula.fv \ \varphi' \subseteq \{0..<b\}) \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MAgg \ g0 \ y \ \omega \ b \ f \ \varphi) \ (Formula.Agg \ y \ \omega \ b \ f \ \varphi')$

$| \text{Prev: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$
 $first \longleftrightarrow j = 0 \implies$
 $list_all2 \ (\lambda i. \ qtable \ n \ (Formula.fv \ \varphi') \ (mem_restr \ R) \ (\lambda v. \ Formula.sat \ \sigma \ V \ (map \ the \ v) \ i \ \varphi'))$
 $\quad [min \ (progress \ \sigma \ P \ \varphi' \ j) \ (j-1)..<progress \ \sigma \ P \ \varphi' \ j] \ buf \implies$
 $list_all2 \ (\lambda i \ t. \ t = \tau \ \sigma \ i) \ [min \ (progress \ \sigma \ P \ \varphi' \ j) \ (j-1)..<j] \ nts \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MPrev \ I \ \varphi \ first \ buf \ nts) \ (Formula.Prev \ I \ \varphi')$

$| \text{Next: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$
 $first \longleftrightarrow progress \ \sigma \ P \ \varphi' \ j = 0 \implies$
 $list_all2 \ (\lambda i \ t. \ t = \tau \ \sigma \ i) \ [progress \ \sigma \ P \ \varphi' \ j - 1..<j] \ nts \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MNext \ I \ \varphi \ first \ nts) \ (Formula.Next \ I \ \varphi')$

$| \text{Since: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \psi \ \psi' \implies$
 $if \ args_pos \ args \ then \ \varphi'' = \varphi' \ else \ \varphi'' = Formula.Neg \ \varphi' \implies$
 $safe_formula \ \varphi'' = args_pos \ args \implies$
 $args_ivl \ args = I \implies$
 $args_n \ args = n \implies$
 $args_L \ args = Formula.fv \ \varphi' \implies$
 $args_R \ args = Formula.fv \ \psi' \implies$
 $Formula.fv \ \varphi' \subseteq Formula.fv \ \psi' \implies$
 $wf_mbuf2' \ \sigma \ P \ V \ j \ n \ R \ \varphi' \ \psi' \ buf \implies$
 $wf_ts \ \sigma \ P \ j \ \varphi' \ \psi' \ nts \implies$
 $wf_since_aux \ \sigma \ V \ R \ args \ \varphi' \ \psi' \ aux \ (progress \ \sigma \ P \ (Formula.Since \ \varphi'' \ I \ \psi') \ j) \implies$
 $wf_mformula \ \sigma \ j \ P \ V \ n \ R \ (MSince \ args \ \varphi \ \psi \ buf \ nts \ aux) \ (Formula.Since \ \varphi'' \ I \ \psi')$

$| \text{Until: } wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies wf_mformula \ \sigma \ j \ P \ V \ n \ R \ \psi \ \psi' \implies$

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

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

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

6.6.2 Initialisation

lemma $wf_mbuf2'_0: pred_mapping\ (\lambda x.\ x = 0)\ P \implies wf_mbuf2'\ \sigma\ P\ V\ 0\ n\ R\ \varphi\ \psi\ ([],\ [])$
unfolding $wf_mbuf2'_def\ wf_mbuf2_def$ **by** *simp*

lemma $wf_mbufn'_0: to_mregex\ r = (mr,\ \varphi s) \implies pred_mapping\ (\lambda x.\ x = 0)\ P \implies wf_mbufn'\ \sigma\ P$
 $V\ 0\ n\ R\ r\ (replicate\ (length\ \varphi s)\ [])$
unfolding $wf_mbufn'_def\ wf_mbufn_def\ map_replicate_const[symmetric]$
by (*auto simp: list_all3_map intro: list_all3_refl simp: Min_eq_iff progress_regex_def*)

lemma $wf_ts_0: wf_ts\ \sigma\ P\ 0\ \varphi\ \psi\ []$
unfolding wf_ts_def **by** *simp*

lemma $wf_ts_regex_0: wf_ts_regex\ \sigma\ P\ 0\ r\ []$
unfolding $wf_ts_regex_def$ **by** *simp*

lemma (in *msaux*) $wf_since_aux_Nil: Formula.fv\ \varphi' \subseteq Formula.fv\ \psi' \implies$
 $wf_since_aux\ \sigma\ V\ R\ (init_args\ I\ n\ (Formula.fv\ \varphi')\ (Formula.fv\ \psi')\ b)\ \varphi'\ \psi'\ (init_msaux\ (init_args\ I$
 $n\ (Formula.fv\ \varphi')\ (Formula.fv\ \psi')\ b))\ 0$
unfolding $wf_since_aux_def$ **by** (*auto intro!: valid_init_msaux*)

lemma (in *muaux*) $wf_until_aux_Nil: Formula.fv\ \varphi' \subseteq Formula.fv\ \psi' \implies$

wf_until_aux σ V R (*init_args* I n (*Formula.fv* φ') (*Formula.fv* ψ') b) φ' ψ' (*init_muaux* (*init_args* I n (*Formula.fv* φ') (*Formula.fv* ψ') b)) 0
unfolding *wf_until_aux_def* *wf_until_auxlist_def* **by** (*auto intro: valid_init_muaux*)

lemma *wf_matchP_aux_Nil*: *wf_matchP_aux* σ V n R I r [] 0
unfolding *wf_matchP_aux_def* **by** *simp*

lemma *wf_matchF_aux_Nil*: *wf_matchF_aux* σ V n R I r [] 0 k
unfolding *wf_matchF_aux_def* **by** *simp*

lemma *fv_regex_alt*: *safe_regex* m g r \implies *Formula.fv_regex* r = $(\bigcup \varphi \in \text{atms } r. \text{Formula.fv } \varphi)$
unfolding *fv_regex_alt* *atms_def*
by (*auto 0 3 dest: safe_regex_safe_formula*)

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

lemma (**in** *maux*) *wf_init0*: *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 (*init0* n φ) φ

proof (*induction arbitrary: n R P V rule: safe_formula_induct*)

case (*Eq_Const* c d)

then show *?case*

by (*auto simp add: is_simple_eq_def simp del: eq_rel.simps intro!: wf_mformula.Eq*)

next

case (*Eq_Var1* c x)

then show *?case*

by (*auto simp add: is_simple_eq_def simp del: eq_rel.simps intro!: wf_mformula.Eq*)

next

case (*Eq_Var2* c x)

then show *?case*

by (*auto simp add: is_simple_eq_def simp del: eq_rel.simps intro!: wf_mformula.Eq*)

next

case (*neq_Var* x y)

then show *?case* **by** (*auto intro!: wf_mformula.neq_Var*)

next

case (*Pred* e ts)

then show *?case* **by** (*auto intro!: wf_mformula.Pred*)

next

case (*Let* p φ ψ)

with *fvi_less_nfv* **show** *?case*

by (*auto simp: pred_mapping_alt_dom_def intro!: wf_mformula.Let Let(4,5)*)

next

case (*And_assign* φ ψ)

then have $1: \forall x \in \text{fv } \psi. x < n$ **by** *simp*

from 1 $\langle \text{safe_assignment } (\text{fv } \varphi) \psi \rangle$

obtain x t **where**

$x < n$ $x \notin \text{fv } \varphi$ $\text{fv_trm } t \subseteq \text{fv } \varphi$

$\psi = \text{Formula.Eq } (\text{Formula.Var } x) t \vee \psi = \text{Formula.Eq } t (\text{Formula.Var } x)$

unfolding *safe_assignment_def* **by** (*force split: formula.splits trm.splits*)

with *And_assign* **show** *?case*

by (*auto intro!: wf_mformula.AndAssign split: trm.splits*)

next

case (*And_safe* φ ψ)

then show *?case* **by** (*auto intro!: wf_mformula.And wf_mbuf2'_0*)

next

case (*And_constraint* φ ψ)

from $\langle \text{fv } \psi \subseteq \text{fv } \varphi \rangle$ $\langle \text{is_constraint } \psi \rangle$

```

obtain  $t1\ p\ c\ t2$  where
  ( $t1, p, c, t2$ ) = split_constraint  $\psi$ 
  formula_of_constraint (split_constraint  $\psi$ ) =  $\psi$ 
   $fv\_trm\ t1 \cup fv\_trm\ t2 \subseteq fv\ \varphi$ 
  by (induction rule: is_constraint.induct) auto
with And_constraint show  $?case$ 
  by (auto 0 3 intro!: wf_mformula.AndRel)
next
case (And_Not  $\varphi\ \psi$ )
then show  $?case$  by (auto intro!: wf_mformula.And wf_mbuf2'_0)
next
case (Ands  $l\ pos\ neg$ )
note  $posneg = Ands.hyps(1)$ 
let  $?wf\_minit = \lambda x. wf\_mformula\ \sigma\ 0\ P\ V\ n\ R\ (minit0\ n\ x)$ 
let  $?pos = filter\ safe\_formula\ l$ 
let  $?neg = filter\ (Not\ \circ\ safe\_formula)\ l$ 
have  $list\_all2\ ?wf\_minit\ ?pos\ pos$ 
  using Ands.IH(1) Ands.prems  $posneg$  by (auto simp: list_all_iff intro!: list.rel_refl_strong)
moreover have  $list\_all2\ ?wf\_minit\ (map\ remove\_neg\ ?neg)\ (map\ remove\_neg\ neg)$ 
  using Ands.IH(2) Ands.prems  $posneg$  by (auto simp: list.rel_map list_all_iff intro!: list.rel_refl_strong)
moreover have  $list\_all3\ (\lambda\ \_ \_ \_.\ True)\ (?pos\ @\ map\ remove\_neg\ ?neg)\ (?pos\ @\ map\ remove\_neg\ ?neg)\ l$ 
  by (auto simp: list_all3_conv_all_nth comp_def sum_length_filter_compl)
moreover have  $l \neq [] \implies (MIN\ \varphi \in set\ l.\ (0 :: nat)) = 0$ 
  by (cases  $l$ ) (auto simp: Min_eq_iff)
ultimately show  $?case$  using Ands.hyps Ands.prems(2)
  by (auto simp: wf_mbufn_def list_all3_map list.rel_map map_replicate_const[symmetric] subset_eq map_map[symmetric] map_append[symmetric] simp del: map_map map_append intro!: wf_mformula.Ands list_all2_appendI)
next
case (Neg  $\varphi$ )
then show  $?case$  by (auto intro!: wf_mformula.Neg)
next
case (Or  $\varphi\ \psi$ )
then show  $?case$  by (auto intro!: wf_mformula.Or wf_mbuf2'_0)
next
case (Exists  $\varphi$ )
then show  $?case$  by (auto simp: fvi_Suc_bound intro!: wf_mformula.Exists)
next
case (Agg  $y\ \omega\ b\ f\ \varphi$ )
then show  $?case$  by (auto intro!: wf_mformula.Agg Agg.IH fvi_plus_bound)
next
case (Prev  $I\ \varphi$ )
thm wf_mformula.Prev[where  $P=P$ ]
then show  $?case$  by (auto intro!: wf_mformula.Prev)
next
case (Next  $I\ \varphi$ )
then show  $?case$  by (auto intro!: wf_mformula.Next)
next
case (Since  $\varphi\ I\ \psi$ )
then show  $?case$ 
  using wf_since_aux_Nil
  by (auto simp add: init_args_def intro!: wf_mformula.Since wf_mbuf2'_0 wf_ts_0)
next
case (Not_Since  $\varphi\ I\ \psi$ )
then show  $?case$ 
  using wf_since_aux_Nil
  by (auto simp add: init_args_def intro!: wf_mformula.Since wf_mbuf2'_0 wf_ts_0)

```

```

next
  case (Until  $\varphi$  I  $\psi$ )
  then show ?case
    using valid_length_muaux[OF valid_init_muaux[OF Until(1)]] wf_until_aux_Nil
    by (auto simp add: init_args_def simp del: progress_simps intro!: wf_mformula.Until wf_mbuf2'_0 wf_ts_0)
next
  case (Not_Until  $\varphi$  I  $\psi$ )
  then show ?case
    using valid_length_muaux[OF valid_init_muaux[OF Not_Until(1)]] wf_until_aux_Nil
    by (auto simp add: init_args_def simp del: progress_simps intro!: wf_mformula.Until wf_mbuf2'_0 wf_ts_0)
next
  case (MatchP I r)
  then show ?case
    by (auto simp: list.rel_map fv_regex_alt simp del: progress_simps split: prod.split
      intro!: wf_mformula.MatchP list.rel_refl_strong wf_mbufn'_0 wf_ts_regex_0 wf_matchP_aux_Nil
      dest!: to_mregex_atms)
next
  case (MatchF I r)
  then show ?case
    by (auto simp: list.rel_map fv_regex_alt progress_le Min_eq_iff progress_regex_def
      simp del: progress_simps split: prod.split
      intro!: wf_mformula.MatchF list.rel_refl_strong wf_mbufn'_0 wf_ts_regex_0 wf_matchF_aux_Nil
      dest!: to_mregex_atms)
qed

```

```

lemma (in mauX) wf_mstate_minit: safe_formula  $\varphi \implies wf\_mstate \varphi \text{ pnil } R (minit \varphi)$ 
  unfolding wf_mstate_def minit_def Let_def
  by (auto intro!: wf_minit0 fvi_less_nfv)

```

6.6.3 Evaluation

```

lemma match_wf_tuple: Some f = match ts xs  $\implies$ 
  wf_tuple n ( $\bigcup t \in \text{set } ts. \text{Formula.fv\_trm } t$ ) (Table.tabulate f 0 n)
  by (induction ts xs arbitrary: f rule: match.induct)
  (fastforce simp: wf_tuple_def split: if_splits option.splits)+

```

```

lemma match_fvi_trm_None: Some f = match ts xs  $\implies \forall t \in \text{set } ts. x \notin \text{Formula.fv\_trm } t \implies f x = \text{None}$ 
  by (induction ts xs arbitrary: f rule: match.induct) (auto split: if_splits option.splits)

```

```

lemma match_fvi_trm_Some: Some f = match ts xs  $\implies t \in \text{set } ts \implies x \in \text{Formula.fv\_trm } t \implies f x \neq \text{None}$ 
  by (induction ts xs arbitrary: f rule: match.induct) (auto split: if_splits option.splits)

```

```

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

```

```

proof (induction ts xs arbitrary: f rule: match.induct)
  case (3 x ts y ys)
  from 3(1)[symmetric] 3(2,3) show ?case
  by (auto 0 3 dest: match_fvi_trm_Some sym split: option.splits if_splits intro!: eval_trm_fv_cong)
qed (auto split: if_splits)

```

```

lemma wf_tuple_tabulate_Some: wf_tuple n A (Table.tabulate f 0 n)  $\implies x \in A \implies x < n \implies \exists y. f x = \text{Some } y$ 
  unfolding wf_tuple_def by auto

```

```

lemma ex_match: wf_tuple n ( $\bigcup t \in \text{set } ts. \text{Formula.fv\_trm } t$ ) v  $\implies$ 

```

$\forall t \in \text{set } ts. (\forall x \in \text{Formula.fv_trm } t. x < n) \wedge (\text{Formula.is_Var } t \vee \text{Formula.is_Const } t) \implies$
 $\exists f. \text{match } ts \text{ (map (Formula.eval_trm (map the } v)) \text{ } ts) = \text{Some } f \wedge v = \text{Table.tabulate } f \ 0 \ n$
proof (induction ts map (Formula.eval_trm (map the v)) ts arbitrary: v rule: match.induct)
case ($\exists x \ ts \ y \ ys$)
then show ?case
proof (cases $x \in (\bigcup t \in \text{set } ts. \text{Formula.fv_trm } t)$)
case True
with \exists **show** ?thesis
by (auto simp: insert_absorb dest!: wf_tuple_tabulate_Some meta_spec[of _ v])
next
case False
with $\exists(3,4)$ **have**
 $*$: map (Formula.eval_trm (map the v)) $ts = \text{map (Formula.eval_trm (map the (} v[x := \text{None}])) \text{ } ts$
by (auto simp: wf_tuple_def nth_list_update intro!: eval_trm_fv_cong)
from False $\exists(2-4)$ **obtain** f **where**
 $\text{match } ts \text{ (map (Formula.eval_trm (map the } v)) \text{ } ts) = \text{Some } f \ v[x := \text{None}] = \text{Table.tabulate } f \ 0 \ n$
unfolding *
by (atomize_elim, intro $\exists(1)$ [of $v[x := \text{None}]$])
(auto simp: wf_tuple_def nth_list_update intro!: eval_trm_fv_cong)
moreover from False **this have** $f \ x = \text{None} \ \text{length } v = n$
by (auto dest: match_fvi_trm_None[OF sym] arg_cong[of _ _ length])
ultimately show ?thesis **using** $\exists(3)$
by (auto simp: list_eq_iff_nth_eq wf_tuple_def)
qed
qed (auto simp: wf_tuple_def intro: nth_equalityI)

lemma eq_rel_eval_trm: $v \in \text{eq_rel } n \ t1 \ t2 \implies \text{is_simple_eq } t1 \ t2 \implies$
 $\forall x \in \text{Formula.fv_trm } t1 \cup \text{Formula.fv_trm } t2. x < n \implies$
 $\text{Formula.eval_trm (map the } v) \ t1 = \text{Formula.eval_trm (map the } v) \ t2$
by (cases $t1$; cases $t2$) (simp_all add: is_simple_eq_def singleton_table_def split: if_splits)

lemma in_eq_rel: $\text{wf_tuple } n \ (\text{Formula.fv_trm } t1 \cup \text{Formula.fv_trm } t2) \ v \implies$
 $\text{is_simple_eq } t1 \ t2 \implies$
 $\text{Formula.eval_trm (map the } v) \ t1 = \text{Formula.eval_trm (map the } v) \ t2 \implies$
 $v \in \text{eq_rel } n \ t1 \ t2$
by (cases $t1$; cases $t2$)
(auto simp: is_simple_eq_def singleton_table_def wf_tuple_def unit_table_def
intro!: nth_equalityI split: if_splits)

lemma table_eq_rel: $\text{is_simple_eq } t1 \ t2 \implies$
 $\text{table } n \ (\text{Formula.fv_trm } t1 \cup \text{Formula.fv_trm } t2) \ (\text{eq_rel } n \ t1 \ t2)$
by (cases $t1$; cases $t2$; simp add: is_simple_eq_def)

lemma wf_tuple_Suc_fviD: $\text{wf_tuple } (\text{Suc } n) \ (\text{Formula.fvi } b \ \varphi) \ v \implies \text{wf_tuple } n \ (\text{Formula.fvi } (\text{Suc } b) \ \varphi) \ (tl \ v)$
unfolding wf_tuple_def **by** (simp add: fvi_Suc_nth_tl)

lemma table_fvi_tl: $\text{table } (\text{Suc } n) \ (\text{Formula.fvi } b \ \varphi) \ X \implies \text{table } n \ (\text{Formula.fvi } (\text{Suc } b) \ \varphi) \ (tl \ 'X)$
unfolding table_def **by** (auto intro: wf_tuple_Suc_fviD)

lemma wf_tuple_Suc_fvi_SomeI: $0 \in \text{Formula.fvi } b \ \varphi \implies \text{wf_tuple } n \ (\text{Formula.fvi } (\text{Suc } b) \ \varphi) \ v \implies$
 $\text{wf_tuple } (\text{Suc } n) \ (\text{Formula.fvi } b \ \varphi) \ (\text{Some } x \ \# \ v)$
unfolding wf_tuple_def
by (auto simp: fvi_Suc_less_Suc_eq_0_disj)

lemma wf_tuple_Suc_fvi_NoneI: $0 \notin \text{Formula.fvi } b \ \varphi \implies \text{wf_tuple } n \ (\text{Formula.fvi } (\text{Suc } b) \ \varphi) \ v \implies$
 $\text{wf_tuple } (\text{Suc } n) \ (\text{Formula.fvi } b \ \varphi) \ (\text{None } \# \ v)$
unfolding wf_tuple_def

by (auto simp: fvi_Suc less_Suc_eq_0_disj)

lemma *qtable_project_fv*: $qtable (Suc\ n) (fv\ \varphi) (mem_restr\ (lift_envs\ R))\ P\ X \implies$
 $qtable\ n\ (Formula.fvi\ (Suc\ 0)\ \varphi)\ (mem_restr\ R)$

($\lambda v. \exists x. P\ ((if\ 0 \in fv\ \varphi\ then\ Some\ x\ else\ None) \# v)$) (tl ' X)

using *neq0_conv* **by** (fastforce simp: image_iff Bex_def fvi_Suc elim!: qtable_cong dest!: qtable_project)

lemma *mem_restr_lift_envs'_append[simp]*:

$length\ xs = b \implies mem_restr\ (lift_envs'\ b\ R)\ (xs\ @\ ys) = mem_restr\ R\ ys$

unfolding *mem_restr_def lift_envs'_def*

by (auto simp: list_all2_append list_rel_map intro!: exI[**where** $x = map\ the\ xs$] list_rel_refl)

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

by (simp add: linorder_not_less list_update_beyond)

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

unfolding *wf_tuple_def*

by (auto simp: nth_list_update_alt)

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

unfolding *mem_restr_def*

by (auto simp: list_all2_conv_all_nth nth_list_update_alt)

lemma *mem_restr_dropI*: $mem_restr\ (lift_envs'\ b\ R)\ xs \implies mem_restr\ R\ (drop\ b\ xs)$

unfolding *mem_restr_def lift_envs'_def*

by (auto simp: append_eq_conv_conj list_all2_append2)

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 –

let $?R = \lambda a\ b. a \neq None \longrightarrow a = Some\ b$

from *assms(2)* **obtain** v **where** $v \in R$ **and** $list_all2\ ?R\ (drop\ b\ xs)\ v$

unfolding *mem_restr_def* ..

show *?thesis* **unfolding** *mem_restr_def* **proof**

have $list_all2\ ?R\ (take\ b\ xs)\ (map\ the\ (take\ b\ xs))$

by (auto simp: list_rel_map intro!: list_rel_refl)

moreover **note** $\langle list_all2\ ?R\ (drop\ b\ xs)\ v \rangle$

ultimately **have** $list_all2\ ?R\ (take\ b\ xs\ @\ drop\ b\ xs)\ (map\ the\ (take\ b\ xs)\ @\ v)$

by (rule *list_all2_appendI*)

then **show** $list_all2\ ?R\ xs\ (map\ the\ (take\ b\ xs)\ @\ v)$ **by** *simp*

show $map\ the\ (take\ b\ xs)\ @\ v \in lift_envs'\ b\ R$

unfolding *lift_envs'_def* **using** *assms(1)* $\langle v \in R \rangle$ **by** *auto*

qed

qed

lemma *wf_tuple_append*: $wf_tuple\ a\ \{x \in A. x < a\}\ xs \implies$

$wf_tuple\ b\ \{x - a \mid x. x \in A \wedge x \geq a\}\ ys \implies$

$wf_tuple\ (a + b)\ A\ (xs\ @\ ys)$

unfolding *wf_tuple_def* **by** (auto simp: nth_append_eq_diff_iff)

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

unfolding *wf_tuple_def* **by** *auto*

lemma *wf_tuple_drop*: $wf_tuple\ (b + n)\ A\ xs \implies \{x - b \mid x. x \in A \wedge x \geq b\} = B \implies$

$wf_tuple\ n\ B\ (drop\ b\ xs)$

unfolding *wf_tuple_def* **by** *force*

lemma *ecard_image*: $\text{inj_on } f \ A \implies \text{ecard } (f \ ' \ A) = \text{ecard } A$
unfolding *ecard_def* **by** (*auto simp: card_image dest: finite_imageD*)

lemma *meval_trm_eval_trm*: $\text{wf_tuple } n \ A \ x \implies \text{fv_trm } t \subseteq A \implies \forall i \in A. i < n \implies$
 $\text{meval_trm } t \ x = \text{Formula.eval_trm } (\text{map the } x) \ t$
unfolding *wf_tuple_def*
by (*induction t simp_all*)

lemma *list_update_id*: $\text{xs } ! \ i = z \implies \text{xs}[i:=z] = \text{xs}$
by (*induction xs arbitrary: i (auto split: nat.split)*)

lemma *qtable_wf_tupleD*: $\text{qtable } n \ A \ P \ Q \ X \implies \forall x \in X. \text{wf_tuple } n \ A \ x$
unfolding *qtable_def table_def* **by** *blast*

lemma *qtable_eval_agg*:
assumes *inner*: $\text{qtable } (b + n) \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } (\text{lift_envs}' \ b \ R))$
 $(\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi) \ \text{rel}$
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 $\text{qtable } n \ (\text{Formula.fv } (\text{Formula.Agg } y \ \omega \ b \ f \ \varphi)) \ (\text{mem_restr } R)$
 $(\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ (\text{Formula.Agg } y \ \omega \ b \ f \ \varphi)) \ (\text{eval_agg } n \ g0 \ y \ \omega \ b \ f \ \text{rel})$
(is *qtable _ ?fv _ ?Q ?rel'*)
proof –
define *M* **where** $M = (\lambda v. \{(x, \text{ecard } Zs) \mid x \ Zs.$
 $Zs = \{zs. \text{length } zs = b \wedge \text{Formula.sat } \sigma \ V \ (zs \ @ \ v) \ i \ \varphi \wedge \text{Formula.eval_trm } (zs \ @ \ v) \ f = x\} \wedge$
 $Zs \neq \{\}\})$
have *f_fvi*: $\text{Formula.fvi_trm } b \ f \subseteq \text{Formula.fvi } b \ \varphi$
using *f_fv* **by** (*auto simp: fvi_trm_iff_fv_trm[where b=b] fvi_iff_fv[where b=b]*)
show *?thesis* **proof** (*cases g0 \wedge rel = empty_table*)
case *True*
then **have** [*simp*]: $\text{Formula.fvi } b \ \varphi = \{\}$
by (*auto simp: g0 fvi_iff_fv(1)[where b=b]*)
then **have** [*simp*]: $\text{Formula.fvi_trm } b \ f = \{\}$
using *f_fvi* **by** *auto*
show *?thesis* **proof** (*rule qtableI*)
show $\text{table } n \ ?fv \ ?rel'$ **by** (*simp add: eval_agg_def True*)
next
fix *v*
assume $\text{wf_tuple } n \ ?fv \ v \ \text{mem_restr } R \ v$
have $\neg \text{Formula.sat } \sigma \ V \ (zs \ @ \ \text{map the } v) \ i \ \varphi$ **if** [*simp*]: $\text{length } zs = b$ **for** *zs*
proof –
let *?zs* = $\text{map2 } (\lambda z \ i. \text{if } i \in \text{Formula.fv } \varphi \ \text{then } \text{Some } z \ \text{else } \text{None}) \ zs \ [0..<b]$
have $\text{wf_tuple } b \ \{x \in \text{fv } \varphi. x < b\} \ ?zs$
by (*simp add: wf_tuple_def*)
then **have** $\text{wf_tuple } (b + n) \ (\text{Formula.fv } \varphi) \ (?zs \ @ \ v[y:=\text{None}])$
using $\langle \text{wf_tuple } n \ ?fv \ v \rangle \ \text{True}$
by (*auto simp: g0 intro!: wf_tuple_append wf_tuple_upd_None*)
then **have** $\neg \text{Formula.sat } \sigma \ V \ (\text{map the } (?zs \ @ \ v[y:=\text{None}])) \ i \ \varphi$
using $\text{True } \langle \text{mem_restr } R \ v \rangle$
by (*auto simp del: map_append dest!: in_qtableI[OF inner, rotated -1]*
intro!: mem_restr_upd_None)
also **have** $\text{Formula.sat } \sigma \ V \ (\text{map the } (?zs \ @ \ v[y:=\text{None}])) \ i \ \varphi \longleftrightarrow \text{Formula.sat } \sigma \ V \ (zs \ @ \ \text{map the } v) \ i \ \varphi$
using True **by** (*auto simp: g0 nth_append intro!: sat_fv_cong*)
finally **show** *?thesis* .

```

qed
then have M_empty: M (map the v) = {}
  unfolding M_def by blast
show Formula.sat  $\sigma$  V (map the v) i (Formula.Agg y  $\omega$  b f  $\varphi$ )
  if v  $\in$  eval_agg n g0 y  $\omega$  b f rel
  using M_empty True that n
  by (simp add: M_def eval_agg_def g0 singleton_table_def)
have v  $\in$  singleton_table n y (the (v ! y)) length v = n
  using <wf_tuple n ?fv v> unfolding wf_tuple_def singleton_table_def
  by (auto simp add: tabulate_alt map_nth
    intro!: trans[OF map_cong[where g=(!) v, simplified nth_map, OF refl], symmetric])
then show v  $\in$  eval_agg n g0 y  $\omega$  b f rel
  if Formula.sat  $\sigma$  V (map the v) i (Formula.Agg y  $\omega$  b f  $\varphi$ )
  using M_empty True that n
  by (simp add: M_def eval_agg_def g0)
qed
next
case non_default_case: False
have union_fv: {0.. $b$ }  $\cup$  ( $\lambda x. x + b$ ) ' Formula.fvi b  $\varphi$  = fv  $\varphi$ 
  using b_fv
  by (auto simp: fvi_iff_fv(1)[where b= $b$ ] intro!: image_eqI[where b= $x$  and  $x=x - b$  for  $x$ ])
have b_n:  $\forall x \in$ fv  $\varphi. x < b + n$ 
proof
  fix x assume x  $\in$  fv  $\varphi$ 
  show x < b + n proof (cases x  $\geq$  b)
    case True
    with <x  $\in$  fv  $\varphi$ > have x - b  $\in$  ?fv
      by (simp add: fvi_iff_fv(1)[where b= $b$ ])
    then show ?thesis using n f_fvi by (auto simp: Un_absorb2)
  qed simp
qed
define M' where M' = ( $\lambda k. \text{let group} = \text{Set.filter } (\lambda x. \text{drop } b \ x = k) \ \text{rel};$ 
  images = meval_trm f ' group
  in ( $\lambda y. (y, \text{ecard } (\text{Set.filter } (\lambda x. \text{meval\_trm } f \ x = y) \ \text{group}))$ ) ' images)
have M'_M: M' (drop b x) = M (map the (drop b x)) if x  $\in$  rel mem_restr (lift_envs' b R) x for x
proof -
  from that have wf_x: wf_tuple (b + n) (fv  $\varphi$ ) x
    by (auto elim!: in_qtableE[OF inner])
  then have wf_zs_x: wf_tuple (b + n) (fv  $\varphi$ ) (map Some zs @ drop b x)
    if length zs = b for zs
    using that b_fv
    by (auto intro!: wf_tuple_append wf_tuple_map_Some wf_tuple_drop)
  have I: (length zs = b  $\wedge$  Formula.sat  $\sigma$  V (zs @ map the (drop b x)) i  $\varphi$   $\wedge$ 
    Formula.eval_trm (zs @ map the (drop b x)) f = y)  $\longleftrightarrow$ 
    ( $\exists a. a \in$  rel  $\wedge$  take b a = map Some zs  $\wedge$  drop b a = drop b x  $\wedge$  meval_trm f a = y)
    (is ?A  $\longleftrightarrow$  ( $\exists a. ?B \ a$ )) for y zs
  proof (intro iffI conjI)
    assume ?A
    then have ?B (map Some zs @ drop (length zs) x)
      using in_qtableI[OF inner wf_zs_x] <mem_restr (lift_envs' b R) x>
      meval_trm_eval_trm[OF wf_zs_x f_fv b_n]
      by (auto intro!: mem_restr_dropI)
    then show  $\exists a. ?B \ a \ ..$ 
  next
    assume  $\exists a. ?B \ a$ 
    then obtain a where ?B a ..
    then have a  $\in$  rel and a_eq: a = map Some zs @ drop b x

```

```

    using append_take_drop_id[of b a] by auto
  then have length a = b + n
    using inner_unfolding qtable_def table_def
    by (blast intro!: wf_tuple_length)
  then show length zs = b
    using wf_tuple_length[OF wf_x] unfolding a_eq by simp
  then have mem_restr (lift_envs' b R) a
    using ⟨mem_restr _ x⟩ unfolding a_eq by (auto intro!: mem_restr_dropI)
  then show Formula.sat σ V (zs @ map the (drop b x)) i φ
    using in_qtableE[OF inner ⟨a ∈ rel⟩]
    by (auto simp: a_eq sat_fv_cong[THEN iffD1, rotated -1])
  from ⟨?B a⟩ show Formula.eval_trm (zs @ map the (drop b x)) f = y
    using meval_trm_eval_trm[OF wf_zs_x f_fv b_n, OF ⟨length zs = b⟩]
    unfolding a_eq by simp
qed
have 2: map Some (map the (take b a)) = take b a if a ∈ rel for a
  using that b_fv inner[THEN qtable_wf_tupleD]
  unfolding table_def wf_tuple_def
  by (auto simp: list_eq_iff_nth_eq)
have 3: ecard {zs. ∃ a. a ∈ rel ∧ take b a = map Some zs ∧ drop b a = drop b x ∧ P a} =
  ecard {a. a ∈ rel ∧ drop b a = drop b x ∧ P a} (is ecard ?A = ecard ?B) for P
proof -
  have ecard ?A = ecard ((λzs. map Some zs @ drop b x) ‘ ?A)
    by (auto intro!: ecard_image[symmetric] inj_onI)
  also have (λzs. map Some zs @ drop b x) ‘ ?A = ?B
    by (subst (1 2) eq_commute) (auto simp: image_iff, metis 2 append_take_drop_id)
  finally show ?thesis .
qed
show ?thesis
  unfolding M_def M'_def
  by (auto simp: non_default_case Let_def image_def 1 3, metis 2)
qed
have drop_lift: mem_restr (lift_envs' b R) x if x ∈ rel mem_restr R ((drop b x)[y:=z]) for x z
proof -
  have (drop b x)[y:=None] = (drop b x)[y:=drop b x ! y] proof -
    from ⟨x ∈ rel⟩ have drop b x ! y = None
      using fresh n inner[THEN qtable_wf_tupleD]
      by (simp add: add_commute wf_tuple_def)
    then show ?thesis by simp
  qed
  then have (drop b x)[y:=None] = drop b x by simp
  moreover from ⟨x ∈ rel⟩ have length x = b + n
    using inner[THEN qtable_wf_tupleD]
    by (simp add: wf_tuple_def)
  moreover from that(2) have mem_restr R ((drop b x)[y:=z, y:=None])
    by (rule mem_restr_upd_None)
  ultimately show ?thesis
    by (auto intro!: mem_restr_dropD)
qed
{
  fix v
  assume mem_restr R v
  have v ∈ (λk. k[y:=Some (eval_agg_op ω (M' k))]) ‘ drop b ‘ rel ⟷
    v ∈ (λk. k[y:=Some (eval_agg_op ω (M (map the k)))] ‘ drop b ‘ rel)
    (is v ∈ ?A ⟷ v ∈ ?B)
proof
  assume v ∈ ?A

```

```

then obtain v' where *: v' ∈ rel v = (drop b v')[y:=Some (eval_agg_op ω (M' (drop b v')))]
  by auto
then have M' (drop b v') = M (map the (drop b v'))
  using ⟨mem_restr R v⟩ by (auto intro!: M'_M drop_lift)
with * show v ∈ ?B by simp
next
assume v ∈ ?B
then obtain v' where *: v' ∈ rel v = (drop b v')[y:=Some (eval_agg_op ω (M (map the (drop b
v'))))]
  by auto
then have M (map the (drop b v')) = M' (drop b v')
  using ⟨mem_restr R v⟩ by (auto intro!: M'_M[symmetric] drop_lift)
with * show v ∈ ?A by simp
qed
then have v ∈ eval_agg n g0 y ω b f rel ↔ v ∈ (λk. k[y:=Some (eval_agg_op ω (M (map the
k)))] ' drop b ' rel
  by (simp add: non_default_case eval_agg_def M'_def Let_def)
}
note alt = this

show ?thesis proof (rule qtableI)
  show table n ?fv ?rel'
    using inner[THEN qtable_wf_tupleD] n f_fvi
    by (auto simp: eval_agg_def non_default_case table_def wf_tuple_def Let_def nth_list_update
      fvi_iff_fv[where b=b] add commute)
  next
    fix v
    assume wf_tuple n ?fv v mem_restr R v
    then have length_v: length v = n by (simp add: wf_tuple_def)

  show Formula.sat σ V (map the v) i (Formula.Agg y ω b f φ)
    if v ∈ eval_agg n g0 y ω b f rel
    proof -
      from that obtain v' where v' ∈ rel
        v = (drop b v')[y:=Some (eval_agg_op ω (M (map the (drop b v'))))]
        using alt[OF ⟨mem_restr R v⟩] by blast
      then have length_v': length v' = b + n
        using inner[THEN qtable_wf_tupleD]
        by (simp add: wf_tuple_def)
      have Formula.sat σ V (map the v') i φ
        using ⟨v' ∈ rel⟩ ⟨mem_restr R v⟩
        by (auto simp: ⟨v = _⟩ elim!: in_qtableE[OF inner] intro!: drop_lift ⟨v' ∈ rel⟩)
      then have Formula.sat σ V (map the (take b v') @ map the v) i φ
      proof (rule sat_fv_cong[THEN iffD1, rotated], intro ballI)
        fix x
        assume x ∈ fv φ
        then have x ≠ y + b using fresh by blast
        moreover have x < length v'
          using ⟨x ∈ fv φ⟩ b_n by (simp add: length_v')
        ultimately show map the v' ! x = (map the (take b v') @ map the v) ! x
          by (auto simp: ⟨v = _⟩ nth_append)
      qed
      then have 1: M (map the v) ≠ {} by (force simp: M_def length_v')

  have y < length (drop b v') using n by (simp add: length_v')
  moreover have Formula.sat σ V (zs @ map the v) i φ ↔
    Formula.sat σ V (zs @ map the (drop b v')) i φ if length zs = b for zs
  proof (intro sat_fv_cong ballI)

```

```

fix x
assume x ∈ fv φ
then have x ≠ y + b using fresh by blast
moreover have x < length v'
  using ⟨x ∈ fv φ⟩ b_n by (simp add: length_v')
ultimately show (zs @ map the v) ! x = (zs @ map the (drop b v')) ! x
  by (auto simp: ⟨v = _⟩ that nth_append)
qed
moreover have Formula.eval_trm (zs @ map the v) f =
  Formula.eval_trm (zs @ map the (drop b v')) f if length zs = b for zs
proof (intro eval_trm_fv_cong ballI)
fix x
assume x ∈ fv_trm f
then have x ≠ y + b using f_fv fresh by blast
moreover have x < length v'
  using ⟨x ∈ fv_trm f⟩ f_fv b_n by (auto simp: length_v')
ultimately show (zs @ map the v) ! x = (zs @ map the (drop b v')) ! x
  by (auto simp: ⟨v = _⟩ that nth_append)
qed
ultimately have map the v ! y = eval_agg_op ω (M (map the v))
  by (simp add: M_def ⟨v = _⟩ conj_commute cong: conj_cong)
with 1 show ?thesis by (auto simp: M_def)
qed

show v ∈ eval_agg n g0 y ω b f rel
if sat_Agg: Formula.sat σ V (map the v) i (Formula.Agg y ω b f φ)
proof -
obtain zs where length zs = b and map Some zs @ v[y:=None] ∈ rel
proof (cases fv φ ⊆ {0..

```

```

      simp: ⟨length zs = b⟩ set_eq_iff fvi_iff_fv[where b=b] fvi_trm_iff_fv_trm[where b=b])
    force+
  with that ⟨length zs = b⟩ show thesis by blast
qed
then have 1: v[y:=None] ∈ drop b ‘ rel by (intro image_eqI) auto

have y_length: y < length v using n by (simp add: length_v)
moreover have Formula.sat σ V (zs @ map the (v[y:=None])) i φ ↔
  Formula.sat σ V (zs @ map the v) i φ if length zs = b for zs
proof (intro sat_fv_cong ballI)
  fix x
  assume x ∈ fv φ
  then have x ≠ y + b using fresh by blast
  moreover have x < b + length v
    using ⟨x ∈ fv φ⟩ b_n by (simp add: length_v)
  ultimately show (zs @ map the (v[y:=None])) ! x = (zs @ map the v) ! x
    by (auto simp: that_nth_append)
qed
moreover have Formula.eval_trm (zs @ map the (v[y:=None])) f =
  Formula.eval_trm (zs @ map the v) f if length zs = b for zs
proof (intro eval_trm_fv_cong ballI)
  fix x
  assume x ∈ fv_trm f
  then have x ≠ y + b using f_fv fresh by blast
  moreover have x < b + length v
    using ⟨x ∈ fv_trm f⟩ f_fv b_n by (auto simp: length_v)
  ultimately show (zs @ map the (v[y:=None])) ! x = (zs @ map the v) ! x
    by (auto simp: that_nth_append)
qed
ultimately have map the v ! y = eval_agg_op ω (M (map the (v[y:=None])))
  using sat_Agg by (simp add: M_def cong: conj_cong) (simp cong: rev_conj_cong)
then have 2: v ! y = Some (eval_agg_op ω (M (map the (v[y:=None])))
  using ⟨wf_tuple n ?fv v⟩ y_length by (auto simp add: wf_tuple_def)
show ?thesis
  unfolding alt[OF ⟨mem_restr R v⟩]
  by (rule image_eqI[where x=v[y:=None]]) (use 1 2 in ⟨auto simp: y_length list_update_id⟩)
qed
qed
qed
qed

```

```

lemma mprev: mprev_next I xs ts = (ys, xs', ts') ⇒
  list_all2 P [i..<j'] xs ⇒ list_all2 (λi t. t = τ σ i) [i..<j] ts ⇒ i ≤ j' ⇒ i < j ⇒
  list_all2 (λi X. if mem (τ σ (Suc i) - τ σ i) I then P i X else X = empty_table)
  [i..<min j' (j-1)] ys ∧
  list_all2 P [min j' (j-1)..<j'] xs' ∧
  list_all2 (λi t. t = τ σ i) [min j' (j-1)..<j] ts'
proof (induction I xs ts arbitrary: i ys xs' ts' rule: mprev_next.induct)
  case (1 I ts)
  then have min j' (j-1) = i by auto
  with 1 show ?case by auto
next
  case (3 I v v' t)
  then have min j' (j-1) = i by (auto simp: list_all2_Cons2 upt_eq_Cons_conv)
  with 3 show ?case by auto
next
  case (4 I x xs t t' ts)
  from 4(1)[of tl ys xs' ts' Suc i] 4(2-6) show ?case

```

by (auto simp add: list_all2_Cons2 upt_eq_Cons_conv Suc_less_eq2
elim!: list.rel_mono_strong split: prod.splits if_splits)
qed simp

lemma mnext: $mprev_next\ I\ xs\ ts = (ys, xs', ts') \implies$
 $list_all2\ P\ [Suc\ i..<j']\ xs \implies list_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [i..<j]\ ts \implies Suc\ i \leq j' \implies i < j \implies$
 $list_all2\ (\lambda i\ X.\ if\ mem\ (\tau\ \sigma\ (Suc\ i) - \tau\ \sigma\ i)\ I\ then\ P\ (Suc\ i)\ X\ else\ X = empty_table)$
 $[i..<min\ (j'-1)\ (j-1)]\ ys \wedge$
 $list_all2\ P\ [Suc\ (min\ (j'-1)\ (j-1))..<j']\ xs' \wedge$
 $list_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [min\ (j'-1)\ (j-1)..<j]\ ts'$

proof (induction I xs ts arbitrary: i ys xs' ts' rule: mprev_next.induct)

case (1 I ts)

then have $min\ (j' - 1)\ (j - 1) = i$ by auto

with 1 show ?case by auto

next

case (3 I v v' t)

then have $min\ (j' - 1)\ (j - 1) = i$ by (auto simp: list_all2_Cons2 upt_eq_Cons_conv)

with 3 show ?case by auto

next

case (4 I x xs t t' ts)

from 4(1)[of tl ys xs' ts' Suc i] 4(2-6) show ?case

by (auto simp add: list_all2_Cons2 upt_eq_Cons_conv Suc_less_eq2
elim!: list.rel_mono_strong split: prod.splits if_splits)

qed simp

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

by (induction xs) auto

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

by (induction xs) auto

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

by (rule sat_fv_cong) (auto intro!: map_the_restrict)

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

by (rule eps_fv_cong) (auto intro!: map_the_restrict)

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

by (induct xs) auto

lemma concat_map_filter[simp]:

$concat\ (map\ f\ (filter\ P\ xs)) = concat\ (map\ (\lambda x.\ if\ P\ x\ then\ f\ x\ else\ [])\ xs)$

by (induct xs) auto

lemma map_filter_alt:

$map\ f\ (filter\ P\ xs) = concat\ (map\ (\lambda x.\ if\ P\ x\ then\ [f\ x]\ else\ [])\ xs)$

by (induct xs) auto

lemma (in maux) update_since:

assumes pre: $wf_since_aux\ \sigma\ V\ R\ args\ \varphi\ \psi\ aux\ ne$

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

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

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

and fvi_subset: $Formula.fv\ \varphi \subseteq Formula.fv\ \psi$

and args_ivl: $args_ivl\ args = I$

and args_n: $args_n\ args = n$

```

and args_L: args_L args = Formula.fv  $\varphi$ 
and args_R: args_R args = Formula.fv  $\psi$ 
and args_pos: args_pos args = pos
shows wf_since_aux  $\sigma$  V R args  $\varphi$   $\psi$  aux' (Suc ne)
and qtable n (Formula.fv  $\psi$ ) (mem_restr R) ( $\lambda v$ . Formula.sat  $\sigma$  V (map the v) ne (Sincep pos  $\varphi$  I
 $\psi$ )) rel
proof -
let ?wf_tuple =  $\lambda v$ . wf_tuple n (Formula.fv  $\psi$ ) v
note sat.simps[simp del]
from pre[unfolded wf_since_aux_def] obtain cur auxlist where aux: valid_msaux args cur aux auxlist
sorted_wrt ( $\lambda x y$ . fst y < fst x) auxlist
 $\wedge t X$ . (t, X)  $\in$  set auxlist  $\implies$  ne  $\neq$  0  $\wedge$  t  $\leq$   $\tau$   $\sigma$  (ne - 1)  $\wedge$   $\tau$   $\sigma$  (ne - 1) - t  $\leq$  right I  $\wedge$ 
( $\exists i$ .  $\tau$   $\sigma$  i = t)  $\wedge$ 
qtable n (fv  $\psi$ ) (mem_restr R)
( $\lambda v$ . Formula.sat  $\sigma$  V (map the v) (ne - 1) (Sincep pos  $\varphi$  (point ( $\tau$   $\sigma$  (ne - 1) - t))  $\psi$ )) X
 $\wedge t$ . ne  $\neq$  0  $\implies$  t  $\leq$   $\tau$   $\sigma$  (ne - 1)  $\implies$   $\tau$   $\sigma$  (ne - 1) - t  $\leq$  right I  $\implies$  ( $\exists i$ .  $\tau$   $\sigma$  i = t)  $\implies$ 
( $\exists X$ . (t, X)  $\in$  set auxlist)
and cur_def:
cur = (if ne = 0 then 0 else  $\tau$   $\sigma$  (ne - 1))
unfolding args_ivl args_n args_pos by blast
from pre[unfolded wf_since_aux_def] have fv_sub: Formula.fv  $\varphi$   $\subseteq$  Formula.fv  $\psi$  by simp

define aux0 where aux0 = join_msaux args rel1 (add_new_ts_msaux args ( $\tau$   $\sigma$  ne) aux)
define auxlist0 where auxlist0 = [(t, join rel pos rel1). (t, rel)  $\leftarrow$  auxlist,  $\tau$   $\sigma$  ne - t  $\leq$  right I]
have tabL: table (args_n args) (args_L args) rel1
using qtable1[unfolded qtable_def] unfolding args_n[symmetric] args_L[symmetric] by simp
have cur_le: cur  $\leq$   $\tau$   $\sigma$  ne
unfolding cur_def by auto
have valid0: valid_msaux args ( $\tau$   $\sigma$  ne) aux0 auxlist0 unfolding aux0_def auxlist0_def
using valid_join_msaux[OF valid_add_new_ts_msaux[OF aux(1)], OF cur_le tabL]
by (auto simp: args_ivl args_pos cur_def map_filter_alt split_beta cong: map_cong)
from aux(2) have sorted_auxlist0: sorted_wrt ( $\lambda x y$ . fst x > fst y) auxlist0
unfolding auxlist0_def
by (induction auxlist) (auto simp add: sorted_wrt_append)
have in_auxlist0_1: (t, X)  $\in$  set auxlist0  $\implies$  ne  $\neq$  0  $\wedge$  t  $\leq$   $\tau$   $\sigma$  (ne-1)  $\wedge$   $\tau$   $\sigma$  ne - t  $\leq$  right I  $\wedge$ 
( $\exists i$ .  $\tau$   $\sigma$  i = t)  $\wedge$ 
qtable n (Formula.fv  $\psi$ ) (mem_restr R) ( $\lambda v$ . (Formula.sat  $\sigma$  V (map the v) (ne-1) (Sincep pos  $\varphi$ 
(point ( $\tau$   $\sigma$  (ne-1) - t))  $\psi$ )  $\wedge$ 
(if pos then Formula.sat  $\sigma$  V (map the v) ne  $\varphi$  else  $\neg$  Formula.sat  $\sigma$  V (map the v) ne  $\varphi$ ))) X for
t X
unfolding auxlist0_def using fvi_subset
by (auto 0 1 elim!: qtable_join[OF _ qtable1] simp: sat_the_restrict dest!: aux(3))
then have in_auxlist0_le_1: (t, X)  $\in$  set auxlist0  $\implies$  t  $\leq$   $\tau$   $\sigma$  ne for t X
by (meson  $\tau$ _mono diff_le_self le_trans)
have in_auxlist0_2: ne  $\neq$  0  $\implies$  t  $\leq$   $\tau$   $\sigma$  (ne-1)  $\implies$   $\tau$   $\sigma$  ne - t  $\leq$  right I  $\implies$   $\exists i$ .  $\tau$   $\sigma$  i = t  $\implies$ 
 $\exists X$ . (t, X)  $\in$  set auxlist0 for t
proof -
fix t
assume ne  $\neq$  0 t  $\leq$   $\tau$   $\sigma$  (ne-1)  $\wedge$   $\tau$   $\sigma$  ne - t  $\leq$  right I  $\exists i$ .  $\tau$   $\sigma$  i = t
then obtain X where (t, X)  $\in$  set auxlist
by (atomize_elim, intro aux(4))
(auto simp: gr0_conv_Suc elim!: order_trans[rotated] intro!: diff_le_mono  $\tau$ _mono)
with  $\langle \tau$   $\sigma$  ne - t  $\leq$  right I  $\rangle$  have (t, join X pos rel1)  $\in$  set auxlist0
unfolding auxlist0_def by (auto elim!: beX[rotated] intro!: exI[of _ X])
then show  $\exists X$ . (t, X)  $\in$  set auxlist0
by blast
qed
have auxlist0_Nil: auxlist0 = []  $\implies$  ne = 0  $\vee$  ne  $\neq$  0  $\wedge$  ( $\forall t$ . t  $\leq$   $\tau$   $\sigma$  (ne-1)  $\wedge$   $\tau$   $\sigma$  ne - t  $\leq$  right I

```

```

→
  (∄ i. τ σ i = t))
  using in_auxlist0_2 by (auto)

have aux'_eq: aux' = add_new_table_msaux args rel2 aux0
  using result_eq unfolding aux0_def update_since_def Let_def by simp
define auxlist' where
  auxlist'_eq: auxlist' = (case auxlist0 of
    [] ⇒ [(τ σ ne, rel2)]
  | x # auxlist' ⇒ (if fst x = τ σ ne then (fst x, snd x ∪ rel2) # auxlist' else (τ σ ne, rel2) # x #
auxlist'))
have tabR: table (args_n args) (args_R args) rel2
  using qtable2[unfolded qtable_def] unfolding args_n[symmetric] args_R[symmetric] by simp
have valid': valid_msaux args (τ σ ne) aux' auxlist'
  unfolding aux'_eq auxlist'_eq using valid_add_new_table_msaux[OF valid0 tabR]
  by (auto simp: not_le split: list.splits option.splits if_splits)
have sorted_auxlist': sorted_wrt (λx y. fst x > fst y) auxlist'
  unfolding auxlist'_eq
  using sorted_auxlist0 in_auxlist0_le_τ by (cases auxlist0) fastforce+
have in_auxlist'_1: t ≤ τ σ ne ∧ τ σ ne - t ≤ right I ∧ (∃ i. τ σ i = t) ∧
  qtable n (Formula.fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) ne (Since pos φ (point
(τ σ ne - t)) ψ)) X
  if auxlist': (t, X) ∈ set auxlist' for t X
proof (cases auxlist0)
  case Nil
  with auxlist' show ?thesis
  unfolding auxlist'_eq using qtable2 auxlist0_Nil
  by (auto simp: zero_enat_def[symmetric] sat_Since_rec[where i=ne]
    dest: spec[of _ τ σ (ne-1)] elim!: qtable_cong[OF _ refl])
next
  case (Cons a as)
  show ?thesis
  proof (cases t = τ σ ne)
    case t: True
    show ?thesis
    proof (cases fst a = τ σ ne)
      case True
      with auxlist' Cons t have X = snd a ∪ rel2
        unfolding auxlist'_eq using sorted_auxlist0 by (auto split: if_splits)
      moreover from in_auxlist0_1[of fst a snd a] Cons have ne ≠ 0
        fst a ≤ τ σ (ne - 1) τ σ ne - fst a ≤ right I ∃ i. τ σ i = fst a
        qtable n (fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) (ne - 1)
          (Since pos φ (point (τ σ (ne - 1) - fst a)) ψ) ∧ (if pos then Formula.sat σ V (map the v)
ne φ
          else ¬ Formula.sat σ V (map the v) ne φ)) (snd a)
        by (auto simp: True[symmetric] zero_enat_def[symmetric])
      ultimately show ?thesis using qtable2 t True
        by (auto simp: sat_Since_rec[where i=ne] sat.simps(6) elim!: qtable_union)
    next
      case False
      with auxlist' Cons t have X = rel2
        unfolding auxlist'_eq using sorted_auxlist0 in_auxlist0_le_τ[of fst a snd a] by (auto split:
if_splits)
      with auxlist' Cons t False show ?thesis
        unfolding auxlist'_eq using qtable2 in_auxlist0_2[of τ σ (ne-1)] in_auxlist0_le_τ[of fst a
snd a] sorted_auxlist0
        by (auto simp: sat_Since_rec[where i=ne] sat.simps(3) zero_enat_def[symmetric] enat_0_iff
not_le

```

```

      elim!: qtable_cong[OF _ refl] dest!: le_τ_less meta_mp)
    qed
  next
  case False
  with auxlist' Cons have (t, X) ∈ set auxlist0
    unfolding auxlist'_eq by (auto split: if_splits)
  then have ne ≠ 0 t ≤ τ σ (ne - 1) τ σ ne - t ≤ right I ∃ i. τ σ i = t
    qtable n (fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) (ne - 1) (Sincep pos φ (point
(τ σ (ne - 1) - t)) ψ) ∧
      (if pos then Formula.sat σ V (map the v) ne φ else ¬ Formula.sat σ V (map the v) ne φ)) X
    using in_auxlist0_1 by blast+
  with False auxlist' Cons show ?thesis
    unfolding auxlist'_eq using qtable2
    by (fastforce simp: sat_Since_rec[where i=ne] sat.simps(6)
      diff_diff_right[where i=τ σ ne and j=τ σ _ + τ σ ne and k=τ σ (ne - 1),
      OF trans_le_add2, simplified] elim!: qtable_cong[OF _ refl] order_trans dest: le_τ_less)
  qed
qed

have in_auxlist'_2: ∃ X. (t, X) ∈ set auxlist' if t ≤ τ σ ne τ σ ne - t ≤ right I ∃ i. τ σ i = t for t
proof (cases t = τ σ ne)
  case True
  then show ?thesis
  proof (cases auxlist0)
    case Nil
    with True show ?thesis unfolding auxlist'_eq by (simp add: zero_enat_def[symmetric])
  next
  case (Cons a as)
  with True show ?thesis unfolding auxlist'_eq
    by (cases fst a = τ σ ne) (auto simp: zero_enat_def[symmetric])
  qed
next
case False
with that have ne ≠ 0
  using le_τ_less neq0_conv by blast
moreover from False that have t ≤ τ σ (ne-1)
  by (metis One_nat_def Suc_leI Suc_pred τ_mono diff_is_0_eq' order.antisym neq0_conv not_le)
ultimately obtain X where (t, X) ∈ set auxlist0 using ⟨τ σ ne - t ≤ right I⟩ ⟨∃ i. τ σ i = t⟩
  using τ_mono[of ne - 1 ne σ] by (atomize_elim, cases right I) (auto intro!: in_auxlist0_2 simp
del: τ_mono)
  then show ?thesis unfolding auxlist'_eq using False ⟨τ σ ne - t ≤ right I⟩
    by (auto intro: exI[of _ X] split: list.split)
  qed

show wf_since_aux σ V R args φ ψ aux' (Suc ne)
  unfolding wf_since_aux_def args_ivl args_n args_pos
  by (auto simp add: fv_sub dest: in_auxlist'_1 intro: sorted_auxlist' in_auxlist'_2
    intro!: exI[of _ auxlist'] valid')

have rel = result_msaux args aux'
  using result_eq by (auto simp add: update_since_def Let_def)
with valid' have rel_eq: rel = foldr (∪) [rel. (t, rel) ← auxlist', left I ≤ τ σ ne - t] {}
  by (auto simp add: args_ivl valid_result_msaux
    intro!: arg_cong[where f = λx. foldr (∪) (concat x) {}] split: option.splits)
have rel_alt: rel = (∪ (t, rel) ∈ set auxlist'. if left I ≤ τ σ ne - t then rel else empty_table)
  unfolding rel_eq
  by (auto elim!: in_foldr_UnE beexI[rotated] intro!: in_foldr_UnI)
show qtable n (fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) ne (Sincep pos φ I ψ)) rel

```

unfolding *rel_alt*
proof (*rule* *qtable_Union*[**where** $Qi=\lambda(t, X) v.$
left $I \leq \tau \sigma ne - t \wedge \text{Formula.sat } \sigma V (\text{map the } v) ne (\text{Sincep pos } \varphi (\text{point } (\tau \sigma ne - t)) \psi)$],
goal_cases *finite* *qtable* *equiv*)
case (*equiv* *v*)
show *?case*
proof (*rule* *iffI*, *erule* *sat_Since_point*, *goal_cases* *left* *right*)
case (*left* *j*)
then show *?case* **using** *in_auxlist'_2*[*of* $\tau \sigma j$, *OF* exI , *OF* $refl$] **by** *auto*
next
case *right*
then show *?case* **by** (*auto* *elim!*: *sat_Since_pointD* *dest*: *in_auxlist'_1*)
qed
qed (*auto* *dest!*: *in_auxlist'_1* *intro!*: *qtable_empty*)
qed

lemma *fv_regex_from_mregex*:
ok (*length* φs) *mr* \implies *fv_regex* (*from_mregex* *mr* φs) $\subseteq (\bigcup \varphi \in \text{set } \varphi s. \text{fv } \varphi)$
by (*induct* *mr*) (*auto* *simp*: *Bex_def* *in_set_conv_nth*) $+$

lemma *qtable_ε_lax*:
assumes *ok* (*length* φs) *mr*
and *list_all2* ($\lambda \varphi \text{ rel. } \text{qtable } n (\text{Formula.fv } \varphi) (\text{mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) i$
 $\varphi) \text{ rel}$) φs *rels*
and *fv_regex* (*from_mregex* *mr* φs) $\subseteq A$ **and** *qtable* *n* *A* (*mem_restr* *R*) *Q* *guard*
shows *qtable* *n* *A* (*mem_restr* *R*)
 $(\lambda v. \text{Regex.eps } (\text{Formula.sat } \sigma V (\text{map the } v)) i (\text{from_mregex } mr \varphi s) \wedge Q v) (\varepsilon_lax \text{ guard } \text{rels } mr)$
using *assms*
proof (*induct* *mr*)
case (*MPlus* *mr1* *mr2*)
from *MPlus*(3-6) **show** *?case*
by (*auto* *intro!*: *qtable_union*[*OF* *MPlus*(1,2)])
next
case (*MTimes* *mr1* *mr2*)
then have *fv_regex* (*from_mregex* *mr1* φs) $\subseteq A$ *fv_regex* (*from_mregex* *mr2* φs) $\subseteq A$
using *fv_regex_from_mregex*[*of* φs *mr1*] *fv_regex_from_mregex*[*of* φs *mr2*] **by** (*auto* *simp*: *sub-*
set_eq)
with *MTimes*(3-6) **show** *?case*
by (*auto* *simp*: *eps_the_restrict* *restrict_idle* *intro!*: *qtable_join*[*OF* *MTimes*(1,2)])
qed (*auto* *split*: *prod_splits* *if_splits* *simp*: *qtable_empty_iff* *list_all2_conv_all_nth*
in_set_conv_nth *restrict_idle* *sat_the_restrict*
intro: *in_qtableI* *qtableI* *elim!*: *qtable_join*[**where** $A=A$ **and** $C=A$])

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

lemma *qtable_empty_unit_table*:
qtable *n* $\{\}$ *R* *P* *empty_table* \implies *qtable* *n* $\{\}$ *R* $(\lambda v. \neg P v)$ (*unit_table* *n*)
by (*auto* *intro*: *qtable_unit_table* *simp* *add*: *qtable_empty_iff*)

lemma *qtable_unit_empty_table*:
qtable *n* $\{\}$ *R* *P* (*unit_table* *n*) \implies *qtable* *n* $\{\}$ *R* $(\lambda v. \neg P v)$ *empty_table*
by (*auto* *intro!*: *qtable_empty* *elim*: *in_qtableE* *simp* *add*: *wf_tuple_empty_unit_table_def*)

lemma *qtable_nonempty_empty_table*:
qtable *n* $\{\}$ *R* *P* *X* $\implies x \in X \implies$ *qtable* *n* $\{\}$ *R* $(\lambda v. \neg P v)$ *empty_table*
by (*frule* *nullary_qtable_cases*) (*auto* *dest*: *qtable_unit_empty_table*)

```

lemma qtable_rε_strict:
  assumes safe_regex Past Strict (from_mregex mr φs) ok (length φs) mr A = fv_regex (from_mregex mr φs)
  and list_all2 (λφ rel. qtable n (Formula.fv φ) (mem_restr R) (λv. Formula.sat σ V (map the v) i φ) rel) φs rels
  shows qtable n A (mem_restr R) (λv. Regex.eps (Formula.sat σ V (map the v)) i (from_mregex mr φs)) (rε_strict n rels mr)
  using assms
proof (hypsubst, induct Past Strict from_mregex mr φs arbitrary: mr rule: safe_regex_induct)
  case (Skip n)
  then show ?case
    by (cases mr) (auto simp: qtable_empty_iff qtable_unit_table split: if_splits)
next
  case (Test φ)
  then show ?case
    by (cases mr) (auto simp: list_all2_conv_all_nth qtable_empty_unit_table dest!: qtable_nonempty_empty_table split: if_splits)
next
  case (Plus r s)
  then show ?case
    by (cases mr) (fastforce intro: qtable_union split: if_splits)+
next
  case (TimesP r s)
  then show ?case
    by (cases mr) (auto intro: qtable_cong[OF qtable_ε_lax] split: if_splits)+
next
  case (Star r)
  then show ?case
    by (cases mr) (auto simp: qtable_unit_table split: if_splits)
qed

```

```

lemma qtable_lε_strict:
  assumes safe_regex Futu Strict (from_mregex mr φs) ok (length φs) mr A = fv_regex (from_mregex mr φs)
  and list_all2 (λφ rel. qtable n (Formula.fv φ) (mem_restr R) (λv. Formula.sat σ V (map the v) i φ) rel) φs rels
  shows qtable n A (mem_restr R) (λv. Regex.eps (Formula.sat σ V (map the v)) i (from_mregex mr φs)) (lε_strict n rels mr)
  using assms
proof (hypsubst, induct Futu Strict from_mregex mr φs arbitrary: mr rule: safe_regex_induct)
  case (Skip n)
  then show ?case
    by (cases mr) (auto simp: qtable_empty_iff qtable_unit_table split: if_splits)
next
  case (Test φ)
  then show ?case
    by (cases mr) (auto simp: list_all2_conv_all_nth qtable_empty_unit_table dest!: qtable_nonempty_empty_table split: if_splits)
next
  case (Plus r s)
  then show ?case
    by (cases mr) (fastforce intro: qtable_union split: if_splits)+
next
  case (TimesF r s)
  then show ?case
    by (cases mr) (auto intro: qtable_cong[OF qtable_ε_lax] split: if_splits)+
next

```

```

    case (Star r)
  then show ?case
    by (cases mr) (auto simp: qtable_unit_table split: if_splits)
qed

```

```

lemma rtranclp_False:  $(\lambda i j. \text{False})^{**} = (=)$ 
proof -
  have  $(\lambda i j. \text{False})^{**} i j \implies i = j$  for  $i j :: 'a$ 
    by (induct i j rule: rtranclp.induct) auto
  then show ?thesis
    by (auto intro: exI[of _ 0])
qed

```

inductive ok_rctxt for φs where

```

  ok_rctxt  $\varphi s$  id id
| ok_rctxt  $\varphi s \kappa \kappa' \implies ok\_rctxt \varphi s (\lambda t. \kappa (MTimes mr t)) (\lambda t. \kappa' (Regex.Times (from_mregex mr \varphi s) t))$ 

```

```

lemma ok_rctxt_swap:  $ok\_rctxt \varphi s \kappa \kappa' \implies from\_mregex (\kappa mr) \varphi s = \kappa' (from\_mregex mr \varphi s)$ 
  by (induct  $\kappa \kappa'$  arbitrary: mr rule: ok_rctxt.induct) auto

```

```

lemma ok_rctxt_cong:  $ok\_rctxt \varphi s \kappa \kappa' \implies Regex.match (Formula.sat \sigma V v) r = Regex.match (Formula.sat \sigma V v) s \implies$ 
   $Regex.match (Formula.sat \sigma V v) (\kappa' r) i j = Regex.match (Formula.sat \sigma V v) (\kappa' s) i j$ 
  by (induct  $\kappa \kappa'$  arbitrary: r s rule: ok_rctxt.induct) simp_all

```

lemma $qtable_rd\kappa$:

```

  assumes ok (length  $\varphi s$ ) mr fv_regex (from_mregex mr  $\varphi s$ )  $\subseteq A$ 
    and list_all2  $(\lambda \varphi rel. qtable n (Formula.fv \varphi) (mem\_restr R) (\lambda v. Formula.sat \sigma V (map the v) j) rel) \varphi s$  rels
    and ok_rctxt  $\varphi s \kappa \kappa'$ 
    and  $\forall ms \in \kappa' RPD mr. qtable n A (mem\_restr R) (\lambda v. Q (map the v) (from_mregex ms \varphi s)) (lookup rel ms)$ 
  shows  $qtable n A (mem\_restr R)$ 
     $(\lambda v. \exists s \in Regex.rpd\kappa \kappa' (Formula.sat \sigma V (map the v)) j (from_mregex mr \varphi s). Q (map the v) s)$ 
     $(rd \kappa rel rels mr)$ 
  using assms

```

proof (induct mr arbitrary: $\kappa \kappa'$)

```

  case MSkip
  then show ?case
    by (auto simp: rtranclp_False ok_rctxt_swap qtable_empty_iff
      elim!: qtable_cong[OF _ _ ok_rctxt_cong[of _  $\kappa \kappa'$ ]] split: nat_splits)

```

next

```

  case (MPlus mr1 mr2)
  from MPlus(3-7) show ?case
    by (auto intro!: qtable_union[OF MPlus(1,2)])

```

next

```

  case (MTimes mr1 mr2)
  from MTimes(3-7) show ?case
    by (auto intro!: qtable_union[OF MTimes(2) qtable_ε_lax[OF _ _ MTimes(1)]]
      elim!: ok_rctxt.intros(2) simp: MTimesL_def Ball_def)

```

next

```

  case (MStar mr)
  from MStar(2-6) show ?case
    by (auto intro!: qtable_cong[OF MStar(1)] intro: ok_rctxt.intros simp: MTimesL_def Ball_def)
qed (auto simp: qtable_empty_iff)

```

lemmas $qtable_rd = qtable_rd\kappa[OF _ _ _ ok_rctxt.intros(1), unfolded rpd\kappa_rpd image_id id_apply]$

inductive *ok_lctxt* **for** φs **where**

ok_lctxt φs *id id*
 $| \text{ok_lctxt } \varphi s \ \kappa \ \kappa' \Longrightarrow \text{ok_lctxt } \varphi s \ (\lambda t. \ \kappa \ (M\text{Times } t \ mr)) \ (\lambda t. \ \kappa' \ (\text{Regex.Times } t \ (\text{from_mregex } mr \ \varphi s)))$

lemma *ok_lctxt_swap*: $\text{ok_lctxt } \varphi s \ \kappa \ \kappa' \Longrightarrow \text{from_mregex } (\kappa \ mr) \ \varphi s = \kappa' \ (\text{from_mregex } mr \ \varphi s)$
by (*induct* $\kappa \ \kappa'$ *arbitrary*: *mr* *rule*: *ok_lctxt.induct*) *auto*

lemma *ok_lctxt_cong*: $\text{ok_lctxt } \varphi s \ \kappa \ \kappa' \Longrightarrow \text{Regex.match } (\text{Formula.sat } \sigma \ V \ v) \ r = \text{Regex.match } (\text{Formula.sat } \sigma \ V \ v) \ s \Longrightarrow$

$\text{Regex.match } (\text{Formula.sat } \sigma \ V \ v) \ (\kappa' \ r) \ i \ j = \text{Regex.match } (\text{Formula.sat } \sigma \ V \ v) \ (\kappa' \ s) \ i \ j$

by (*induct* $\kappa \ \kappa'$ *arbitrary*: *r s* *rule*: *ok_lctxt.induct*) *simp_all*

lemma *qtable_ldk*:

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

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

and *ok_lctxt* $\varphi s \ \kappa \ \kappa'$

and $\forall ms \in \kappa' \text{ 'LPD } mr. \ \text{qtable } n \ A \ (\text{mem_restr } R) \ (\lambda v. \ Q \ (\text{map } \text{the } v) \ (\text{from_mregex } ms \ \varphi s)) \ (\text{lookup } \text{rel } ms)$

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

$(\lambda v. \ \exists s \in \text{Regex.lpd} \ \kappa \ \kappa' \ (\text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v)) \ j \ (\text{from_mregex } mr \ \varphi s). \ Q \ (\text{map } \text{the } v) \ s)$

$(\text{ld } \kappa \ \text{rel } \text{rels } mr)$

using *assms*

proof (*induct* *mr* *arbitrary*: $\kappa \ \kappa'$)

case *MSkip*

then show *?case*

by (*auto simp*: *rtranclp_False* *ok_lctxt_swap* *qtable_empty_iff*)

elim!: *qtable_cong*[*OF* *__* *ok_rctxt_cong*[*of* *_* $\kappa \ \kappa'$]] *split*: *nat.splits*)

next

case (*MPlus* *mr1* *mr2*)

from *MPlus*(3-7) **show** *?case*

by (*auto intro!*: *qtable_union*[*OF* *MPlus*(1,2)])

next

case (*MTimes* *mr1* *mr2*)

from *MTimes*(3-7) **show** *?case*

by (*auto intro!*: *qtable_union*[*OF* *MTimes*(1)] *qtable_ε_lax*[*OF* *__* *MTimes*(2)])

elim!: *ok_lctxt.intros*(2) *simp*: *MTimesR_def* *Ball_def*)

next

case (*MStar* *mr*)

from *MStar*(2-6) **show** *?case*

by (*auto intro!*: *qtable_cong*[*OF* *MStar*(1)] *intro*: *ok_lctxt.intros* *simp*: *MTimesR_def* *Ball_def*)

qed (*auto simp*: *qtable_empty_iff*)

lemmas *qtable_ld* = *qtable_ldk*[*OF* *__* *ok_lctxt.intros*(1), *unfolded* *lpdk_lpd* *image_id* *id_apply*]

lemma *RPD_fv_regex_le*:

$ms \in \text{RPD } mr \Longrightarrow \text{fv_regex } (\text{from_mregex } ms \ \varphi s) \subseteq \text{fv_regex } (\text{from_mregex } mr \ \varphi s)$

by (*induct* *mr* *arbitrary*: *ms*) (*auto simp*: *MTimesL_def* *split*: *nat.splits*)**+**

lemma *RPD_safe*: *safe_regex* *Past* *g* (*from_mregex* *mr* φs) \Longrightarrow

$ms \in \text{RPD } mr \Longrightarrow \text{safe_regex } \text{Past } g \ (\text{from_mregex } ms \ \varphi s)$

proof (*induct* *Past* *g* *from_mregex* *mr* φs *arbitrary*: *mr* *ms* *rule*: *safe_regex_induct*)

case *Skip*

then show *?case*

by (*cases* *mr*) (*auto split*: *nat.splits*)

next

case (*Test* *g* φ)

```

then show ?case
  by (cases mr) auto
next
case (Plus g r s mrs)
then show ?case
proof (cases mrs)
  case (MPlus mr ms)
  with Plus(3-5) show ?thesis
  by (auto dest!: Plus(1,2))
qed auto
next
case (TimesP g r s mrs)
then show ?case
proof (cases mrs)
  case (MTimes mr ms)
  with TimesP(3-5) show ?thesis
  by (cases g) (auto 0 4 simp: MTimesL_def dest: RPD_fv_regex_le TimesP(1,2))
qed auto
next
case (Star g r)
then show ?case
proof (cases mr)
  case (MStar x6)
  with Star(2-4) show ?thesis
  by (cases g) (auto 0 4 simp: MTimesL_def dest: RPD_fv_regex_le
    elim!: safe_cosafe[rotated] dest!: Star(1))
qed auto
qed

lemma RPDi_safe: safe_regex Past g (from_mregex mr  $\varphi$ s)  $\implies$ 
  ms  $\in$  RPDi n mr  $\implies$  safe_regex Past g (from_mregex ms  $\varphi$ s)
by (induct n arbitrary: ms mr) (auto dest: RPD_safe)

lemma RPDs_safe: safe_regex Past g (from_mregex mr  $\varphi$ s)  $\implies$ 
  ms  $\in$  RPDs mr  $\implies$  safe_regex Past g (from_mregex ms  $\varphi$ s)
unfolding RPDs_def by (auto dest: RPDi_safe)

lemma RPD_safe_fv_regex: safe_regex Past Strict (from_mregex mr  $\varphi$ s)  $\implies$ 
  ms  $\in$  RPD mr  $\implies$  fv_regex (from_mregex ms  $\varphi$ s) = fv_regex (from_mregex mr  $\varphi$ s)
proof (induct Past Strict from_mregex mr  $\varphi$ s arbitrary: mr rule: safe_regex_induct)
  case (Skip n)
  then show ?case
  by (cases mr) (auto split: nat.splits)
next
case (Test  $\varphi$ )
then show ?case
  by (cases mr) auto
next
case (Plus r s)
then show ?case
  by (cases mr) auto
next
case (TimesP r s)
then show ?case
  by (cases mr) (auto 0 3 simp: MTimesL_def dest: RPD_fv_regex_le split: modality.splits)
next
case (Star r)
then show ?case

```

by (cases mr) (auto 0 3 simp: MTimesL_def dest: RPD_fv_regex_le)
qed

lemma *RPDi_safe_fv_regex*: safe_regex Past Strict (from_mregex mr φ s) \implies
 $ms \in \text{RPDi } n \text{ mr} \implies \text{fv_regex (from_mregex ms } \varphi\text{s)} = \text{fv_regex (from_mregex mr } \varphi\text{s)}$
by (induct n arbitrary: ms mr) (auto 5 0 dest: RPD_safe_fv_regex RPD_safe)

lemma *RPDs_safe_fv_regex*: safe_regex Past Strict (from_mregex mr φ s) \implies
 $ms \in \text{RPDs } mr \implies \text{fv_regex (from_mregex ms } \varphi\text{s)} = \text{fv_regex (from_mregex mr } \varphi\text{s)}$
unfolding *RPDs_def* **by** (auto dest: RPDi_safe_fv_regex)

lemma *RPD_ok*: $ok \ m \ mr \implies ms \in \text{RPD } mr \implies ok \ m \ ms$
proof (induct mr arbitrary: ms)
case (MPlus mr1 mr2)
from *MPlus(3,4)* **show** ?case
by (auto elim: MPlus(1,2))
next
case (MTimes mr1 mr2)
from *MTimes(3,4)* **show** ?case
by (auto elim: MTimes(1,2) simp: MTimesL_def)
next
case (MStar mr)
from *MStar(2,3)* **show** ?case
by (auto elim: MStar(1) simp: MTimesL_def)
qed (auto split: nat.splits)

lemma *RPDi_ok*: $ok \ m \ mr \implies ms \in \text{RPDi } n \text{ mr} \implies ok \ m \ ms$
by (induct n arbitrary: ms mr) (auto intro: RPD_ok)

lemma *RPDs_ok*: $ok \ m \ mr \implies ms \in \text{RPDs } mr \implies ok \ m \ ms$
unfolding *RPDs_def* **by** (auto intro: RPDi_ok)

lemma *LPD_fv_regex_le*:
 $ms \in \text{LPD } mr \implies \text{fv_regex (from_mregex ms } \varphi\text{s)} \subseteq \text{fv_regex (from_mregex mr } \varphi\text{s)}$
by (induct mr arbitrary: ms) (auto simp: MTimesR_def split: nat.splits)+

lemma *LPD_safe*: safe_regex Futu g (from_mregex mr φ s) \implies
 $ms \in \text{LPD } mr \implies \text{safe_regex Futu } g \text{ (from_mregex ms } \varphi\text{s)}$
proof (induct Futu g from_mregex mr φ s arbitrary: mr ms rule: safe_regex_induct)
case Skip
then show ?case
by (cases mr) (auto split: nat.splits)
next
case (Test g φ)
then show ?case
by (cases mr) auto
next
case (Plus g r s mrs)
then show ?case
proof (cases mrs)
case (MPlus mr ms)
with *Plus(3-5)* **show** ?thesis
by (auto dest!: Plus(1,2))
qed auto
next
case (TimesF g r s mrs)
then show ?case
proof (cases mrs)

```

    case (MTimes mr ms)
  with TimesF(3-5) show ?thesis
    by (cases g) (auto 0 4 simp: MTimesR_def dest: LPD_fv_regex_le split: modality.splits dest:
TimesF(1,2))
  qed auto
next
case (Star g r)
then show ?case
proof (cases mr)
  case (MStar x6)
  with Star(2-4) show ?thesis
    by (cases g) (auto 0 4 simp: MTimesR_def dest: LPD_fv_regex_le
elim!: safe_cosafe[rotated] dest!: Star(1))
  qed auto
qed

```

lemma $LPDi_safe$: $safe_regex\ Futu\ g\ (from_mregex\ mr\ \varphi s) \implies ms \in LPDi\ n\ mr \implies safe_regex\ Futu\ g\ (from_mregex\ ms\ \varphi s)$
by (induct n arbitrary: ms mr) (auto dest: LPD_safe)

lemma $LPDs_safe$: $safe_regex\ Futu\ g\ (from_mregex\ mr\ \varphi s) \implies ms \in LPDs\ mr \implies safe_regex\ Futu\ g\ (from_mregex\ ms\ \varphi s)$
unfolding $LPDs_def$ **by** (auto dest: LPDi_safe)

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 (induct Futu Strict from_mregex mr φs arbitrary: mr rule: safe_regex_induct)
 case Skip
 then show ?case
 by (cases mr) (auto split: nat.splits)
next
 case (Test φ)
 then show ?case
 by (cases mr) auto
next
 case (Plus r s)
 then show ?case
 by (cases mr) auto
next
 case (TimesF r s)
 then show ?case
 by (cases mr) (auto 0 3 simp: MTimesR_def dest: LPD_fv_regex_le split: modality.splits)
next
 case (Star r)
 then show ?case
 by (cases mr) (auto 0 3 simp: MTimesR_def dest: LPD_fv_regex_le)
qed

lemma $LPDi_safe_fv_regex$: $safe_regex\ Futu\ Strict\ (from_mregex\ mr\ \varphi s) \implies ms \in LPDi\ n\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
by (induct n arbitrary: ms mr) (auto 5 0 dest: LPD_safe_fv_regex LPD_safe)

lemma $LPDs_safe_fv_regex$: $safe_regex\ Futu\ Strict\ (from_mregex\ mr\ \varphi s) \implies ms \in LPDs\ mr \implies fv_regex\ (from_mregex\ ms\ \varphi s) = fv_regex\ (from_mregex\ mr\ \varphi s)$
unfolding $LPDs_def$ **by** (auto dest: LPDi_safe_fv_regex)

lemma LPD_ok : $ok\ m\ mr \implies ms \in LPD\ mr \implies ok\ m\ ms$
proof (induct mr arbitrary: ms)

```

    case (MPlus mr1 mr2)
  from MPlus(3,4) show ?case
  by (auto elim: MPlus(1,2))
next
  case (MTimes mr1 mr2)
  from MTimes(3,4) show ?case
  by (auto elim: MTimes(1,2) simp: MTimesR_def)
next
  case (MStar mr)
  from MStar(2,3) show ?case
  by (auto elim: MStar(1) simp: MTimesR_def)
qed (auto split: nat.splits)

lemma LPDi_ok: ok m mr  $\implies$  ms  $\in$  LPDi n mr  $\implies$  ok m ms
  by (induct n arbitrary: ms mr) (auto intro: LPD_ok)

lemma LPDs_ok: ok m mr  $\implies$  ms  $\in$  LPDs mr  $\implies$  ok m ms
  unfolding LPDs_def by (auto intro: LPDi_ok)

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 -
  let ?wf_tuple =  $\lambda v$ . wf_tuple n (Formula.fv_regex r) v
  let ?update =  $\lambda$ rel t. mrtabulate mrs ( $\lambda$ mr.
  r  $\delta$  id rel rels mr  $\cup$  (if t =  $\tau$   $\sigma$  ne then r $\varepsilon$ _strict n rels mr else {}))
  note sat.simps[simp del]

  define aux0 where aux0 = [(t, ?update rel t). (t, rel)  $\leftarrow$  aux, enat ( $\tau$   $\sigma$  ne - t)  $\leq$  right I]
  have sorted_aux0: sorted_wrt ( $\lambda x y$ . fst x > fst y) aux0
  using pre[unfolded wf_matchP_aux_def, THEN conjunct1]
  unfolding aux0_def
  by (induction aux) (auto simp add: sorted_wrt_append)
  { fix ms
  assume ms  $\in$  RPDs mr
  then have fv_regex (from_mregex ms  $\varphi$ s) = fv_regex r
  safe_regex Past Strict (from_mregex ms  $\varphi$ s) ok (length  $\varphi$ s) ms RPD ms  $\subseteq$  RPDs mr
  using safe RPDs_safe RPDs_safe_fv_regex mr from_mregex_to_mregex RPDs_ok to_mregex_ok
RPDs_trans
  by fastforce+
  } note * = this
  have **:  $\tau$   $\sigma$  ne - ( $\tau$   $\sigma$  i +  $\tau$   $\sigma$  ne -  $\tau$   $\sigma$  (ne - Suc 0)) =  $\tau$   $\sigma$  (ne - Suc 0) -  $\tau$   $\sigma$  i for i
  by (metis (no_types, lifting) Nat.diff_diff_right  $\tau$ _mono add commute add_diff_cancel_left diff_le_self
le_add2 order_trans)
  have ***:  $\tau$   $\sigma$  i =  $\tau$   $\sigma$  ne
  if  $\tau$   $\sigma$  ne  $\leq$   $\tau$   $\sigma$  i  $\tau$   $\sigma$  i  $\leq$   $\tau$   $\sigma$  (ne - Suc 0) ne > 0 for i
  by (metis (no_types, lifting) Suc_pred  $\tau$ _mono diff_le_self le_ $\tau$ _less le_antisym not_less_eq that)
  then have in_aux0_1: (t, X)  $\in$  set aux0  $\implies$  ne  $\neq$  0  $\wedge$  t  $\leq$   $\tau$   $\sigma$  ne  $\wedge$   $\tau$   $\sigma$  ne - t  $\leq$  right I  $\wedge$ 
  ( $\exists$  i.  $\tau$   $\sigma$  i = t)  $\wedge$ 
  ( $\forall$  ms  $\in$  RPDs mr. qtable n (fv_regex r) (mem_restr R) ( $\lambda v$ . Formula.sat  $\sigma$  V (map the v) ne

```



```

next
case (Cons a as)
show ?thesis
proof (cases t =  $\tau \sigma ne$ )
case t: True
show ?thesis
proof (cases fst a =  $\tau \sigma ne$ )
case True
with aux' Cons t have X = snd a
unfolding aux'_eq using sorted_aux0 by auto
moreover from in_aux0_1[of fst a snd a] Cons have ne  $\neq 0$ 
fst a  $\leq \tau \sigma ne$   $\tau \sigma ne - fst a \leq right I \exists i. \tau \sigma i = fst a$ 
 $\forall ms \in RPDs mr. qtable n (fv\_regex r) (mem\_restr R) (\lambda v. Formula.sat \sigma V (map the v) ne$ 
(Formula.MatchP (point ( $\tau \sigma ne - fst a$ )) (from_mregex ms  $\varphi s$ ))) (lookup (snd a) ms)
by auto
ultimately show ?thesis using t True
by auto
next
case False
with aux' Cons t have X = mrtabulate mrs (r $\varepsilon$ _strict n rels)
unfolding aux'_eq using sorted_aux0 in_aux0_le_ $\tau$ [of fst a snd a] by auto
with aux' Cons t False show ?thesis
unfolding aux'_eq using safe mrs qtables * in_aux0_2[of  $\tau \sigma (ne-1)$ ] in_aux0_le_ $\tau$ [of fst a
snd a] sorted_aux0
by (auto simp: sat_MatchP_rec[where i= $ne$ ] zero_enat_def[symmetric] enat_0_iff not_le
lookup_tabulate finite_RPDs split: option.splits
intro!: qtable_cong[OF qtable_r $\varepsilon$ _strict] dest!: le_ $\tau$ _less meta_mp)
qed
next
case False
with aux' Cons have (t, X)  $\in$  set aux0
unfolding aux'_eq by (auto split: if_splits)
then have ne  $\neq 0$   $t \leq \tau \sigma ne$   $\tau \sigma ne - t \leq right I \exists i. \tau \sigma i = t$ 
 $\forall ms \in RPDs mr. qtable n (fv\_regex r) (mem\_restr R) (\lambda v. Formula.sat \sigma V (map the v) ne$ 
(Formula.MatchP (point ( $\tau \sigma ne - t$ )) (from_mregex ms  $\varphi s$ ))) (lookup X ms)
using in_aux0_1 by blast+
with False aux' Cons show ?thesis
unfolding aux'_eq by auto
qed
qed

have in_aux'_2:  $\exists X. (t, X) \in set aux'$  if  $t \leq \tau \sigma ne$   $\tau \sigma ne - t \leq right I \exists i. \tau \sigma i = t$  for t
proof (cases t =  $\tau \sigma ne$ )
case True
then show ?thesis
proof (cases aux0)
case Nil
with True show ?thesis unfolding aux'_eq by simp
next
case (Cons a as)
with True show ?thesis unfolding aux'_eq using eq_fst_iff[of t a]
by (cases fst a =  $\tau \sigma ne$ ) auto
qed
next
case False
with that have ne  $\neq 0$ 
using le_ $\tau$ _less neq0_conv by blast
moreover from False that have  $t \leq \tau \sigma (ne-1)$ 

```

```

    by (metis One_nat_def Suc_leI Suc_pred  $\tau$ _mono diff_is_0_eq' order.antisym neq0_conv not_le)
  ultimately obtain X where (t, X)  $\in$  set aux0 using  $\langle \tau \sigma ne - t \leq \text{right } I \rangle \langle \exists i. \tau \sigma i = t \rangle$ 
    by atomize_elim (auto intro!: in_aux0_2)
  then show ?thesis unfolding aux'_eq using False
    by (auto intro: exI[of _ X] split: list.split)
qed

show wf_matchP_aux  $\sigma$  V n R I r aux' (Suc ne)
  unfolding wf_matchP_aux_def using mr
  by (auto dest: in_aux'_1 intro: sorted_aux' in_aux'_2)

have rel_eq: rel = foldr ( $\cup$ ) [lookup rel mr. (t, rel)  $\leftarrow$  aux', left I  $\leq$   $\tau \sigma ne - t$ ] {}
  unfolding aux'_eq aux0_def
  using result_eq by (simp add: update_matchP_def Let_def)
have rel_alt: rel = ( $\bigcup$  (t, rel)  $\in$  set aux'. if left I  $\leq$   $\tau \sigma ne - t$  then lookup rel mr else empty_table)
  unfolding rel_eq
  by (auto elim!: in_foldr_UnE bezI[rotated] intro!: in_foldr_UnI)
show qtable n (fv_regex r) (mem_restr R) ( $\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) ne (\text{Formula.MatchP } I r)$ ) rel
  unfolding rel_alt
  proof (rule qtable_Union[where Qi= $\lambda(t, X) v.$ 
    left I  $\leq$   $\tau \sigma ne - t \wedge \text{Formula.sat } \sigma V (\text{map the } v) ne (\text{Formula.MatchP } (\text{point } (\tau \sigma ne - t)) r)$ ],
    goal_cases finite qtable equiv)
  case (equiv v)
  show ?case
  proof (rule iffI, erule sat_MatchP_point, goal_cases left right)
  case (left j)
  then show ?case using in_aux'_2[of  $\tau \sigma j$ , OF __ exI, OF __ refl] by auto
  next
  case right
  then show ?case by (auto elim!: sat_MatchP_pointD dest: in_aux'_1)
  qed
  qed (auto dest!: in_aux'_1 intro!: qtable_empty dest!: bspec[OF __ RPDs_refl]
    simp: from_mregex_eq[OF safe mr])
qed

lemma length_update_until: length (update_until args rel1 rel2 nt aux) = Suc (length aux)
  unfolding update_until_def by simp

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) ( $\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) (ne + \text{length auxlist}) \varphi$ ) rel1
  and qtable2: qtable n (Formula.fv  $\psi$ ) (mem_restr R) ( $\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) (ne + \text{length auxlist}) \psi$ ) rel2
  and fvi_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 ( $\tau \sigma (ne + \text{length auxlist})$ )
    auxlist) ne
  unfolding wf_until_auxlist_def length_update_until
  unfolding update_until_def list.rel_map add_Suc_right upt.simps eqTrueI[OF le_add1] if_True
  proof (rule list_all2_appendI, unfold list.rel_map, goal_cases old new)
  case old
  show ?case
  proof (rule list.rel_mono_strong[OF assms(1)[unfolded wf_until_auxlist_def]]; safe, goal_cases mono1
    mono2)

```

```

case (mono1 i X Y v)
then show ?case
  by (fastforce simp: args_ivl args_n args_pos sat_the_restrict less_Suc_eq
    elim!: qtable_join[OF _ qtable1] qtable_union[OF _ qtable1])
next
case (mono2 i X Y v)
then show ?case using fvi_subset
  by (auto 0 3 simp: args_ivl args_n args_pos sat_the_restrict less_Suc_eq split: if_splits
    elim!: qtable_union[OF _ qtable_join_fixed[OF qtable2]]
    elim: qtable_cong[OF _ refl] intro: exI[of _ ne + length_auxlist])
qed
next
case new
then show ?case
  by (auto intro!: qtable_empty qtable1 qtable2[THEN qtable_cong] exI[of _ ne + length_auxlist]
    simp: args_ivl args_n args_pos less_Suc_eq zero_enat_def[symmetric])
qed

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 +
length_muaux args aux) φ) rel1
    and qtable2: qtable n (Formula.fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) (ne +
length_muaux args aux) ψ) rel2
    and fvi_subset: Formula.fv φ ⊆ 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 + length_muaux args
aux)) aux) ne ∧
    length_muaux args (add_new_muaux args rel1 rel2 (τ σ (ne + length_muaux args aux)) aux) =
Suc (length_muaux args aux)
proof -
from pre obtain cur auxlist where valid_aux: valid_muaux args cur aux auxlist and
  cur: cur = (if ne + length_auxlist = 0 then 0 else τ σ (ne + length_auxlist - 1)) and
  pre_list: wf_until_auxlist σ V n R pos φ I ψ auxlist ne
unfolding wf_until_aux_def args_ivl args_n args_pos by auto
have length_aux: length_muaux args aux = length_auxlist
  using valid_length_muaux[OF valid_aux] .
define nt where nt ≡ τ σ (ne + length_muaux args aux)
have nt_mono: cur ≤ nt
  unfolding cur nt_def length_aux by simp
define auxlist' where auxlist' ≡ update_until args rel1 rel2 (τ σ (ne + length_auxlist)) auxlist
have length_auxlist': length_auxlist' = Suc (length_auxlist)
  unfolding auxlist'_def by (auto simp add: length_update_until)
have tab1: table (args_n args) (args_L args) rel1
  using qtable1 unfolding args_n[symmetric] args_L[symmetric] by (auto simp add: qtable_def)
have tab2: table (args_n args) (args_R args) rel2
  using qtable2 unfolding args_n[symmetric] args_R[symmetric] by (auto simp add: qtable_def)
have fv_sub: fv φ ⊆ fv ψ
  using pre unfolding wf_until_aux_def by auto
moreover have valid_add_new_auxlist: valid_muaux args nt (add_new_muaux args rel1 rel2 nt aux)
auxlist'
  using valid_add_new_muaux[OF valid_aux tab1 tab2 nt_mono]
  unfolding auxlist'_def nt_def length_aux .
moreover have length_muaux args (add_new_muaux args rel1 rel2 nt aux) = Suc (length_muaux args

```

```

aux)
  using valid_length_muaux[OF valid_add_new_auxlist] unfolding length_auxlist' length_aux[symmetric]
  .
  moreover have wf_until_auxlist  $\sigma$  V n R pos  $\varphi$  I  $\psi$  auxlist' ne
    using wf_update_until_auxlist[OF pre_list qtable1[unfolded length_aux] qtable2[unfolded length_aux]
fv_sub args_ivl args_n args_pos]
    unfolding auxlist'_def .
  moreover have  $\tau$   $\sigma$  (ne + length auxlist) = (if ne + length auxlist' = 0 then 0 else  $\tau$   $\sigma$  (ne + length
auxlist' - 1))
    unfolding cur length_auxlist' by auto
  ultimately show ?thesis
    unfolding wf_until_aux_def nt_def length_aux args_ivl args_n args_pos by fast
qed

lemma length_update_matchF_base:
  length (fst (update_matchF_base I mr mrs nt entry st)) = Suc 0
  by (auto simp: Let_def update_matchF_base_def)

lemma length_update_matchF_step:
  length (fst (update_matchF_step I mr mrs nt entry st)) = Suc (length (fst st))
  by (auto simp: Let_def update_matchF_step_def split: prod.splits)

lemma length_foldr_update_matchF_step:
  length (fst (foldr (update_matchF_step I mr mrs nt) aux base)) = length aux + length (fst base)
  by (induct aux arbitrary: base) (auto simp: Let_def length_update_matchF_step)

lemma length_update_matchF: length (update_matchF n I mr mrs rels nt aux) = Suc (length aux)
  unfolding update_matchF_def update_matchF_base_def length_foldr_update_matchF_step
  by (auto simp: Let_def)

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  $\varphi$ ) (mem_restr R) ( $\lambda v$ . Formula.sat  $\sigma$  V (map
the v) j  $\varphi$ ) rel)  $\varphi$ s rels
  shows wf_matchF_invar  $\sigma$  V n R I r (update_matchF_base n I mr mrs rels ( $\tau$   $\sigma$  j)) j
proof -
  have from_mregex: from_mregex mr  $\varphi$ s = r
    using safe mr using from_mregex_eq by blast
  { fix ms
    assume ms  $\in$  LPDs mr
    then have fv_regex (from_mregex ms  $\varphi$ s) = fv_regex r
      safe_regex Futu Strict (from_mregex ms  $\varphi$ s) ok (length  $\varphi$ s) ms LPD ms  $\subseteq$  LPDs mr
      using safe LPDs_safe LPDs_safe_fv_regex mr from_mregex_to_mregex LPDs_ok to_mregex_ok
LPDs_trans
      by fastforce+
    } note * = this
  show ?thesis
    unfolding wf_matchF_invar_def wf_matchF_aux_def update_matchF_base_def mr prod.case Let_def
mrs
    using safe
    by (auto simp: * from_mregex qtables qtable_empty_iff zero_enat_def[symmetric]
lookup_tabulate finite_LPDs eps_match less_Suc_eq LPDs_refl
intro!: qtable_cong[OF qtable_le_strict[where  $\varphi$ s= $\varphi$ s]] intro: qtables exI[of _ j]
split: option.splits)
qed

```

lemma *Un_empty_table[simp]*: $rel \cup empty_table = rel \cup empty_table \cup rel = rel$
unfolding *empty_table_def* **by** *auto*

lemma *wf_matchF_invar_step*:

assumes *wf*: *wf_matchF_invar* $\sigma V n R I r st (Suc i)$
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* $(\lambda \varphi rel. qtable\ n\ (Formula.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi)\ rel)\ \varphi s\ rels$
and *rel*: *qtable* $n\ (Formula.fv_regex\ r)\ (mem_restr\ R)\ (\lambda v. (\exists j. i \leq j \wedge j < i + length\ (fst\ st) \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge Regex.match\ (Formula.sat\ \sigma\ V\ (map\ the\ v))\ r\ i\ j))\ rel$
and *entry*: *entry* $= (\tau\ \sigma\ i, rels, rel)$
and *nt*: *nt* $= \tau\ \sigma\ (i + length\ (fst\ st))$
shows *wf_matchF_invar* $\sigma V n R I r (update_matchF_step\ I\ mr\ mrs\ nt\ entry\ st)\ i$
proof –
have *from_mregex*: *from_mregex* $mr\ \varphi s = r$
using *safe mr using from_mregex_eq* **by** *blast*
{ fix *ms*
assume *ms* $\in LPDs\ mr$
then have *fv_regex* $(from_mregex\ ms\ \varphi s) = fv_regex\ r$
safe_regex Futu Strict (from_mregex ms φs) ok (length φs) ms LPD ms ⊆ LPDs mr
using *safe LPDs_safe LPDs_safe fv_regex mr from_mregex_to_mregex LPDs_ok to_mregex_ok LPDs_trans*
by *fastforce+*
} **note** $*$ $= this$
{ fix *aux X ms*
assume *st* $= (aux, X)\ ms \in LPDs\ mr$
with *wf mr have* *qtable* $n\ (fv_regex\ r)\ (mem_restr\ R)$
 $(\lambda v. Regex.match\ (Formula.sat\ \sigma\ V\ (map\ the\ v))\ (from_mregex\ ms\ \varphi s)\ i\ (i + length\ aux))$
 $(l\delta\ (\lambda x. x)\ X\ rels\ ms)$
by $(intro\ qtable_cong[OF\ qtable_l\delta][where\ \varphi s = \varphi s\ and\ A = fv_regex\ r\ and\ Q = \lambda v\ r. Regex.match\ (Formula.sat\ \sigma\ V\ v)\ r\ (Suc\ i)\ (i + length\ aux),\ OF\ __\ qtables])$
 $(auto\ simp: wf_matchF_invar_def\ * LPDs_trans\ lpd_match[of\ i]\ elim!: bspec)$
} **note** $l\delta = this$
have *lookup* $(mrtabulate\ mrs\ f)\ ms = f\ ms$ **if** $ms \in LPDs\ mr$ **for** ms **and** $f :: mregex \Rightarrow 'a\ table$
using *that mrs by (fastforce simp: lookup_tabulate finite_LPDS split: option.splits)+*
then show *?thesis*
using *wf mr mrs entry nt LPDs_trans*
by $(auto\ 0\ 3\ simp: Let_def\ wf_matchF_invar_def\ update_matchF_step_def\ wf_matchF_aux_def\ mr\ * LPDs_refl$
 $list_all2_Cons1\ append_eq_Cons_conv\ upt_eq_Cons_conv\ Suc_le_eq\ qtables$
 $lookup_tabulate\ finite_LPDs\ id_def\ l\delta\ from_mregex\ less_Suc_eq$
 $intro!: qtable_union[OF\ rel\ l\delta]\ qtable_cong[OF\ rel]$
 $intro: exI[of_ i + length\ _]$
 $split: if_splits\ prod.splits)$

qed

lemma *wf_update_matchF_invar*:

assumes *pre*: *wf_matchF_aux* $\sigma V n R I r aux ne (length\ (fst\ st) - 1)$
and *wf*: *wf_matchF_invar* $\sigma V n R I r st (ne + length\ aux)$
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*: $j = ne + length\ aux + length\ (fst\ st) - 1$
shows *wf_matchF_invar* $\sigma V n R I r (foldr\ (update_matchF_step\ I\ mr\ mrs\ (\tau\ \sigma\ j))\ aux\ st)\ ne$
using *pre wf unfolding j*

proof (*induct aux arbitrary: ne*)
case (*Cons entry aux*)
from *Cons(1)[of Suc ne] Cons(2,3)* **show** *?case*
unfolding *foldr.simps o_apply*
by (*intro wf_matchF_invar_step[where rels = fst (snd entry) and rel = snd (snd entry)]*)
(auto simp: safe mr mrs wf_matchF_aux_def wf_matchF_invar_def list_all2_Cons1 append_eq_Cons_conv
Suc_le_eq upt_eq_Cons_conv length_foldr_update_matchF_step add.assoc split: if_splits)
qed *simp*

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, φ s)*
and *mrs: mrs = sorted_list_of_set (LPDs mr)*
and *nt: nt = τ σ (ne + length aux)*
and *qtables: list_all2 ($\lambda\varphi$ rel. qtable n (Formula.fv φ) (mem_restr R) (λv . Formula.sat σ V (map*
the v) (ne + length aux) φ) rel) φ s rels
shows *wf_matchF_aux σ V n R I r (update_matchF n I mr mrs rels nt aux) ne 0*
unfolding *update_matchF_def using wf_update_matchF_base_invar[OF safe mr mrs qtables, of I]*
unfolding *nt*
by (*intro wf_update_matchF_invar[OF __ safe mr mrs, unfolded wf_matchF_invar_def split_beta,*
THEN conjunct2, THEN conjunct1])
(auto simp: length_update_matchF_base wf_matchF_invar_def update_matchF_base_def Let_def
pre)

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)
unfolding *wf_until_auxlist_def*
by (*simp add: upt_conv_Cons del: upt_Suc cong: if_cong*)

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
unfolding *wf_matchF_aux_def*
by (*simp add: upt_conv_Cons del: upt_Suc cong: if_cong split: prod.splits*)

lemma *wf_until_aux_Cons1*: *wf_until_auxlist σ V n R pos φ I ψ ((t, a1, a2) # aux) ne \implies t = τ σ*
ne
unfolding *wf_until_auxlist_def*
by (*simp add: upt_conv_Cons del: upt_Suc*)

lemma *wf_matchF_aux_Cons1*: *wf_matchF_aux σ V n R I r ((t, rels, rel) # aux) ne i \implies t = τ σ*
ne
unfolding *wf_matchF_aux_def*
by (*simp add: upt_conv_Cons del: upt_Suc split: prod.splits*)

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) (λv . ($\exists j$. ne \leq j \wedge j < Suc (ne + length aux) \wedge mem (τ σ j -
 τ σ ne) I \wedge
Formula.sat σ V (map the v) j ψ \wedge ($\forall k \in \{ne..<j\}$. if pos then Formula.sat σ V (map the v) k φ else
 \neg Formula.sat σ V (map the v) k φ))) a2
unfolding *wf_until_auxlist_def*
by (*simp add: upt_conv_Cons del: upt_Suc*)

lemma *wf_matchF_aux_Cons3*: *wf_matchF_aux σ V n R I r ((t, rels, rel) # aux) ne i \implies*
qtable n (Formula.fv_regex r) (mem_restr R) (λv . ($\exists j$. ne \leq j \wedge j < Suc (ne + length aux + i) \wedge mem
(τ σ j - τ σ ne) I \wedge
Regex.match (Formula.sat σ V (map the v)) r ne j)) rel

unfolding *wf_matchF_aux_def*
by (*simp add: upt_conv_Cons del: upt_Suc split: prod.splits*)

lemma *upt_append*: $a \leq b \implies b \leq c \implies [a..<b] @ [b..<c] = [a..<c]$
by (*metis le_Suc_ex upt_add_eq_append*)

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 $ja \leq ja' \quad jb \leq jb'$
shows *wf_mbuf2 i ja' jb' P Q (mbuf2_add xs ys buf)*
using *assms unfolding wf_mbuf2_def*
by (*auto 0 3 simp: list_all2_append2 upt_append dest: list_all2_lengthD*
intro: exI[where x=[i..<ja]] exI[where x=[ja..<ja']]
exI[where x=[i..<jb]] exI[where x=[jb..<jb']] split: prod.splits)

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)*
unfolding *wf_mbufn_def list_all3_conv_all_nth*
proof *safe*
show $\text{length } Ps = \text{length } js' \quad \text{length } js' = \text{length } (\text{mbufn_add } xss \text{ buf})$
using *assms unfolding wf_mbufn_def list_all3_conv_all_nth list_all2_conv_all_nth* **by** *auto*
next
fix *k* **assume** $k < \text{length } Ps$
then show $i \leq js' ! k$
using *assms unfolding wf_mbufn_def list_all3_conv_all_nth list_all2_conv_all_nth*
by (*auto 0 4 dest: spec[of_ i]*)
from *k* **have** $[i..<js' ! k] = [i..<js ! k] @ [js ! k ..<js' ! k]$ **and**
 $\text{length } [i..<js ! k] = \text{length } (\text{buf} ! k)$
using *assms(1,3) unfolding wf_mbufn_def list_all3_conv_all_nth list_all2_conv_all_nth*
by (*auto simp: upt_append*)
with *k* **show** $\text{list_all2 } (Ps ! k) [i..<js' ! k] (\text{mbufn_add } xss \text{ buf} ! k)$
using *assms[unfolded wf_mbufn_def list_all3_conv_all_nth]*
by (*auto simp add: list_all2_append*)
qed

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*
using *assms unfolding wf_mbuf2_def*
by (*induction f buf arbitrary: i xs buf' rule: mbuf2_take.induct*)
(fastforce simp add: list_all2_Cons2 upt_eq_Cons_conv min_absorb1 min_absorb2 split: prod.splits)+

lemma *list_all3_Nil[simp]*:
 $\text{list_all3 } P \text{ xs ys } [] \longleftrightarrow \text{xs} = [] \wedge \text{ys} = []$
 $\text{list_all3 } P \text{ xs } [] \text{ zs} \longleftrightarrow \text{xs} = [] \wedge \text{zs} = []$
 $\text{list_all3 } P [] \text{ ys zs} \longleftrightarrow \text{ys} = [] \wedge \text{zs} = []$
unfolding *list_all3_conv_all_nth* **by** *auto*

lemma *list_all3_Cons*:
 $\text{list_all3 } P \text{ xs ys } (z \# \text{zs}) \longleftrightarrow (\exists x \text{ xs}' y \text{ ys}'. \text{xs} = x \# \text{xs}' \wedge \text{ys} = y \# \text{ys}' \wedge P x y z \wedge \text{list_all3 } P \text{ xs}' \text{ ys}' \text{ zs})$

$list_all3\ P\ xs\ (y\ \#\ ys)\ zs \longleftrightarrow (\exists\ x\ xs'\ z\ zs'.\ xs = x\ \#\ xs' \wedge zs = z\ \#\ zs' \wedge P\ x\ y\ z \wedge list_all3\ P\ xs'\ ys\ zs')$
 $list_all3\ P\ (x\ \#\ xs)\ ys\ zs \longleftrightarrow (\exists\ y\ ys'\ z\ zs'.\ ys = y\ \#\ ys' \wedge zs = z\ \#\ zs' \wedge P\ x\ y\ z \wedge list_all3\ P\ xs\ ys'\ zs')$

unfolding $list_all3_conv_all_nth$
by (*auto simp: length_Suc_conv Suc_length_conv nth_Cons split: nat.splits*)

lemma $list_all3_mono_strong: list_all3\ P\ xs\ ys\ zs \implies$
 $(\bigwedge x\ y\ z.\ x \in set\ xs \implies y \in set\ ys \implies z \in set\ zs \implies P\ x\ y\ z \implies Q\ x\ y\ z) \implies$
 $list_all3\ Q\ xs\ ys\ zs$
by (*induct xs ys zs rule: list_all3.induct*) (*auto intro: list_all3.intros*)

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 $[] \in set\ buf \implies Mini\ i\ js = i$
unfolding $in_set_conv_nth$

proof (*elim exE conjE*)

fix k

have $i \leq j$ **if** $j \in set\ js$ **for** j

using *that assms* **unfolding** $wf_mbufn_def\ list_all3_conv_all_nth\ in_set_conv_nth$ **by** *auto*
moreover assume $k < length\ buf\ buf\ !\ k = []$

ultimately show *?thesis* **using** *assms*

unfolding $Mini_def\ wf_mbufn_def\ list_all3_conv_all_nth$
by (*auto 0 4 dest!: spec[of _ k] intro: Min_eqI simp: in_set_conv_nth*)

qed

lemma $wf_mbufn_notin_set:$

assumes $wf_mbufn\ i\ js\ Ps\ buf$
shows $[] \notin set\ buf \implies j \in set\ js \implies i < j$
using *assms* **unfolding** $wf_mbufn_def\ list_all3_conv_all_nth$
by (*cases i \in set js*) (*auto intro: le_neq_implies_less simp: in_set_conv_nth*)

lemma $wf_mbufn_map_tl:$

$wf_mbufn\ i\ js\ Ps\ buf \implies [] \notin set\ buf \implies wf_mbufn\ (Suc\ i)\ js\ Ps\ (map\ tl\ buf)$
by (*auto simp: wf_mbufn_def list_all3_map Suc_le_eq*
dest: rel_funD[OF tl_transfer] elim!: list_all3_mono_strong le_neq_implies_less)

lemma $list_all3_list_all2I: list_all3\ (\lambda x\ y\ z.\ Q\ x\ z)\ xs\ ys\ zs \implies list_all2\ Q\ xs\ zs$

by (*induct xs ys zs rule: list_all3.induct*) *auto*

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 $ja \leq j\ jb \leq j$
shows $wf_mbuf2\ (min\ ja\ jb)\ ja\ jb\ P\ Q\ buf'$
and $list_all2\ R\ [min\ ja\ jb..<j]\ nts'$
using *assms* **unfolding** wf_mbuf2_def
by (*induction f z buf nts arbitrary: i z' buf' nts' rule: mbuf2t_take.induct*)
(auto simp add: list_all2_Cons2 upt_eq_Cons_conv less_eq_Suc_le min_absorb1 min_absorb2
split: prod.split)

lemma $wf_mbufn_take:$

assumes $mbufn_take\ f\ z\ buf = (z',\ buf')$
and $wf_mbufn\ i\ js\ Ps\ buf$

```

shows wf_mbufn (Mini i js) js Ps buf'
using assms unfolding wf_mbufn_def
proof (induction f z buf arbitrary: i z' buf' rule: mbufn_take.induct)
case rec: (1 f z buf)
show ?case proof (cases buf = [])
case True
with rec.prem1 show ?thesis by simp
next
case nonempty: False
show ?thesis proof (cases [] ∈ set buf)
case True
from rec.prem2(2) have ∀j∈set js. i ≤ j
by (auto simp: in_set_conv_nth list_all3_conv_all_nth)
moreover from True rec.prem2(2) have i ∈ set js
by (fastforce simp: in_set_conv_nth list_all3_conv_all_nth list_all2_iff)
ultimately have Mini i js = i
unfolding Mini_def
by (auto intro!: antisym[OF Min.coboundedI Min.boundedI])
with rec.prem1 nonempty True show ?thesis by simp
next
case False
from nonempty rec.prem2(2) have Mini i js = Mini (Suc i) js
unfolding Mini_def by auto
show ?thesis
unfolding ‹Mini i js = Mini (Suc i) js›
proof (rule rec.IH)
show ¬ (buf = [] ∨ [] ∈ set buf) using nonempty False by simp
show list_all3 (λP j xs. Suc i ≤ j ∧ list_all2 P [Suc i..<j] xs) Ps js (map tl buf)
using False rec.prem2(2)
by (auto simp: list_all3_map elim!: list_all3_mono_strong dest: list.rel_sel[THEN iffD1])
show mbufn_take f (f (map hd buf) z) (map tl buf) = (z', buf')
using nonempty False rec.prem1 by simp
qed
qed
qed
qed

```

```

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 ∧k. k ∈ set js ⇒ k ≤ j
and k = Mini (i + length nts) js
shows wf_mbufn k js Ps buf'
and list_all2 R [k..<j] nts'
using assms(1-4) unfolding assms(5)
proof (induction f z buf nts arbitrary: i z' buf' nts' rule: mbufnt_take.induct)
case IH: (1 f z buf nts)
note mbufnt_take.simps[simp del]
case 1
then have *: list_all2 R [Suc i..<j] (tl nts)
by (auto simp: list.rel_sel[of R [i..<j] nts, simplified])
from 1 show ?case
using wf_mbufn_in_set_Minim[OF 1(2)]
by (subst (asm) mbufnt_take.simps)
(force simp: Mini_def wf_mbufn_def split: if_splits prod.splits elim!: list_all3_mono_strong
dest!: IH(1)[rotated, OF _ wf_mbufn_map_tl[OF 1(2)] *])
case 2

```

```

then have *: list_all2 R [Suc i..<j] (tl nts)
  by (auto simp: list.rel_sel[of R [i..<j] nts, simplified])
have [simp]: Suc (i + (length nts - Suc 0)) = i + length nts if nts ≠ []
  using that by (fastforce simp flip: length_greater_0_conv)
with 2 show ?case
  using wf_mbufn_in_set_Mini[OF 2(2)] wf_mbufn_notin_set[OF 2(2)]
  by (subst (asm) mbufnt_take.simps) (force simp: Mini_def wf_mbufn_def
    dest!: IH(2)[rotated, OF _ wf_mbufn_map_tl[OF 2(2)] *]
    split: if_splits prod.splits)
qed

lemma mbuf2t_take_induct[consumes 5, case_names base step]:
  assumes mbuf2t_take f z buf nts = (z', buf', nts')
    and wf_mbuf2 i ja jb P Q buf
    and list_all2 R [i..<j] nts
    and ja ≤ j jb ≤ j
    and U i z
    and  $\bigwedge k x y t z. i \leq k \implies \text{Suc } k \leq ja \implies \text{Suc } k \leq jb \implies$ 
       $P k x \implies Q k y \implies R k t \implies U k z \implies U (\text{Suc } k) (f x y t z)$ 
  shows U (min ja jb) z'
  using assms unfolding wf_mbuf2_def
  by (induction f z buf nts arbitrary: i z' buf' nts' rule: mbuf2t_take.induct)
    (auto simp add: list_all2_Cons2 upt_eq_Cons_conv less_eq_Suc_le min_absorb1 min_absorb2
      elim!: arg_cong2[of _ _ _ _ U, OF _ refl, THEN iffD1, rotated] split: prod.split)

lemma list_all2_hdD:
  assumes list_all2 P [i..<j] xs xs ≠ []
  shows P i (hd xs) i < j
  using assms unfolding list_all2_conv_all_nth
  by (auto simp: hd_conv_nth intro: zero_less_diff[THEN iffD1] dest!: spec[of _ 0])

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  $\bigwedge k xs z. i \leq k \implies \text{Suc } k \leq \text{Mini } i \text{ } js \implies$ 
       $\text{list\_all2 } (\lambda P x. P k x) Ps xs \implies U k z \implies U (\text{Suc } k) (f xs z)$ 
  shows U (Mini i js) z'
  using assms unfolding wf_mbufn_def
  proof (induction f z buf arbitrary: i z' buf' rule: mbufn_take.induct)
  case rec: (1 f z buf)
  show ?case proof (cases buf = [])
  case True
  with rec.prem1 show ?thesis unfolding Mini_def by simp
  next
  case nonempty: False
  show ?thesis proof (cases [] ∈ set buf)
  case True
  from rec.prem2(2) have  $\forall j \in \text{set } js. i \leq j$ 
  by (auto simp: in_set_conv_nth list_all3_conv_all_nth)
  moreover from True rec.prem2(2) have  $i \in \text{set } js$ 
  by (fastforce simp: in_set_conv_nth list_all3_conv_all_nth list_all2_iff)
  ultimately have Mini i js = i
  unfolding Mini_def
  by (auto intro!: antisym[OF Min.coboundedI Min.boundedI])
  with rec.prem1 nonempty True show ?thesis by simp
  next
  case False
  
```

```

with nonempty rec.premis(2) have more: Suc i ≤ Mini i js
  using diff_is_0_eq not_le unfolding Mini_def
  by (fastforce simp: in_set_conv_nth list_all3_conv_all_nth list_all2_iff)
then have Mini i js = Mini (Suc i) js unfolding Mini_def by auto
show ?thesis
  unfolding ‹Mini i js = Mini (Suc i) js›
proof (rule rec.IH)
show ¬ (buf = [] ∨ [] ∈ set buf) using nonempty False by simp
show mbufn_take f (f (map hd buf) z) (map tl buf) = (z', buf')
  using nonempty False rec.premis by simp
show list_all3 (λP j xs. Suc i ≤ j ∧ list_all2 P [Suc i..<j] xs) Ps js (map tl buf)
  using False rec.premis
  by (auto simp: list_all3_map elim!: list_all3_mono_strong dest: list.rel_sel[THEN iffD1])
show U (Suc i) (f (map hd buf) z)
  using more False rec.premis
  by (auto 0 4 simp: list_all3_map intro!: rec.premis(4) list_all3_list_all2I
      elim!: list_all3_mono_strong dest: list.rel_sel[THEN iffD1])
show ∧k xs z. Suc i ≤ k ⇒ Suc k ≤ Mini (Suc i) js ⇒
  list_all2 (λP. P k) Ps xs ⇒ U k z ⇒ U (Suc k) (f xs z)
  by (rule rec.premis(4)) (auto simp: Mini_def)
qed
qed
qed
qed

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 ∧k. k ∈ set js ⇒ k ≤ j
    and U i z
    and ∧k xs t z. i ≤ k ⇒ Suc k ≤ Mini j js ⇒
      list_all2 (λP x. P k x) Ps xs ⇒ R k t ⇒ U k z ⇒ U (Suc k) (f xs t z)
  shows U (Mini (i + length nts) js) z'
  using assms
proof (induction f z buf nts arbitrary: i z' buf' nts' rule: mbufnt_take.induct)
case (1 f z buf nts)
then have *: list_all2 R [Suc i..<j] (tl nts)
  by (auto simp: list.rel_sel[of R [i..<j] nts, simplified])
note mbufnt_take.simps[simp del]
from 1(2-6) have i = Min (set js) if js ≠ [] nts = []
  using that unfolding wf_mbufn_def using wf_mbufn_in_set_Mini[OF 1(3)]
  by (fastforce simp: Mini_def list_all3_Cons neq_Nil_conv)
with 1(2-7) list_all2_hdD[OF 1(4)] show ?case
  unfolding wf_mbufn_def using wf_mbufn_in_set_Mini[OF 1(3)] wf_mbufn_notin_set[OF 1(3)]
  by (subst (asm) mbufnt_take.simps)
    (auto simp add: Mini_def list.rel_map Suc_le_eq
      elim!: arg_cong2[of _ _ _ U, OF _ refl, THEN iffD1, rotated]
      list_all3_mono_strong[THEN list_all3_list_all2I[of _ _ js]] list_all2_hdD
      dest!: 1(1)[rotated, OF _ wf_mbufn_map_tl[OF 1(3)] * _ 1(7)] split: prod.split if_splits)
qed

lemma mbuf2_take_add':
  assumes eq: mbuf2_take f (mbuf2_add xs ys buf) = (zs, buf')
    and pre: wf_mbuf2' σ P V j n R φ ψ buf
    and rm: rel_mapping (≤) P P'
    and xs: list_all2 (λt. qtable n (Formula.fv φ) (mem_restr R) (λv. Formula.sat σ V (map the v) i φ))
      [progress σ P φ j..<progress σ P' φ j'] xs

```

and $ys: list_all2 (\lambda i. qtable\ n\ (Formula.fv\ \psi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \psi))$
 $[progress\ \sigma\ P\ \psi\ j..<progress\ \sigma\ P'\ \psi\ j']\ ys$
and $j \leq j'$
shows $wf_mbuf2'\ \sigma\ P'\ V\ j'\ n\ R\ \varphi\ \psi\ buf'$
and $list_all2 (\lambda i\ Z. \exists X\ Y.$
 $qtable\ n\ (Formula.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi)\ X \wedge$
 $qtable\ n\ (Formula.fv\ \psi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \psi)\ Y \wedge$
 $Z = f\ X\ Y)$
 $[min\ (progress\ \sigma\ P\ \varphi\ j)\ (progress\ \sigma\ P'\ \varphi\ j')..<min\ (progress\ \sigma\ P'\ \varphi\ j')\ (progress\ \sigma\ P'\ \psi\ j')]$ zs
using $force\ rm\ unfolding\ wf_mbuf2'_def$
by $(force\ intro!:\ mbuf2_take_eqD[OF\ eq]\ wf_mbuf2_add[OF\ _xs\ ys]\ progress_mono_gen[OF\ \langle j \leq j' \rangle$
 $rm])+$

lemma $mbuf2_take_add'$:

assumes $eq: mbuf2_take\ f\ z\ (mbuf2_add\ xs\ ys\ buf)\ nts = (z', buf', nts')$
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 $xs: list_all2 (\lambda i. qtable\ n\ (Formula.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi))$
 $[progress\ \sigma\ P\ \varphi\ j..<progress\ \sigma\ P'\ \varphi\ j']\ xs$
and $ys: list_all2 (\lambda i. qtable\ n\ (Formula.fv\ \psi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \psi))$
 $[progress\ \sigma\ P\ \psi\ j..<progress\ \sigma\ P'\ \psi\ j']\ ys$
and $j \leq j'$
shows $wf_mbuf2'\ \sigma\ P'\ V\ j'\ n\ R\ \varphi\ \psi\ buf'$
and $wf_ts\ \sigma\ P'\ j'\ \varphi\ \psi\ nts'$
using $pre_buf\ pre_nts\ bounded\ rm\ unfolding\ wf_mbuf2'_def\ wf_ts_def$
by $(auto\ intro!:\ mbuf2_take_eqD[OF\ eq]\ wf_mbuf2_add[OF\ _xs\ ys]\ progress_mono_gen[OF\ \langle j \leq j' \rangle$
 $rm])$
 $progress_le_gen)$

lemma $ok_0_atms: ok\ 0\ mr \implies regex.atms\ (from_mregex\ mr\ []) = \{\}$

by $(induct\ mr)\ auto$

lemma $ok_0_progress: ok\ 0\ mr \implies progress_regex\ \sigma\ P\ (from_mregex\ mr\ [])\ j = j$

by $(drule\ ok_0_atms)\ (auto\ simp: progress_regex_def)$

lemma $atms_empty_atms: safe_regex\ m\ g\ r \implies atms\ r = \{\} \iff regex.atms\ r = \{\}$

by $(induct\ r\ rule: safe_regex_induct)\ (auto\ split: if_splits\ simp: cases_Neg_iff)$

lemma $atms_empty_progress: safe_regex\ m\ g\ r \implies atms\ r = \{\} \implies progress_regex\ \sigma\ P\ r\ j = j$

by $(drule\ atms_empty_atms)\ (auto\ simp: progress_regex_def)$

lemma $to_mregex_empty_progress: safe_regex\ m\ g\ r \implies to_mregex\ r = (mr, []) \implies$

$progress_regex\ \sigma\ P\ r\ j = j$

using $from_mregex_eq\ ok_0_progress\ to_mregex_ok\ atms_empty_atms$ **by** $fastforce$

lemma $progress_regex_le: pred_mapping\ (\lambda x. x \leq j)\ P \implies progress_regex\ \sigma\ P\ r\ j \leq j$

by $(auto\ intro!:\ progress_le_gen\ simp: Min_le_iff\ progress_regex_def)$

lemma $Neg_acyclic: formula.Neg\ x = x \implies P$

by $(induct\ x)\ auto$

lemma $case_Neg_in_iff: x \in (case\ y\ of\ formula.Neg\ \varphi' \Rightarrow \{\varphi'\} \mid _ \Rightarrow \{\}) \iff y = formula.Neg\ x$

by $(cases\ y)\ auto$

lemma $atms_nonempty_progress:$

$safe_regex\ m\ g\ r \implies atms\ r \neq \{\} \implies (\lambda \varphi. progress\ \sigma\ P\ \varphi\ j) \text{ ' } atms\ r = (\lambda \varphi. progress\ \sigma\ P\ \varphi\ j) \text{ ' }$

regex.atms r
by (*frule atms_empty_atms; simp*)
(auto 0 3 simp: atms_def image_iff case_Neg_in_iff elim!: disjE Not2 dest: safe_regex_safe_formula)

lemma *to_mregex_nonempty_progress: safe_regex m g r \implies to_mregex r = (mr, φ s) \implies φ s \neq [] \implies progress_regex σ P r j = (MIN $\varphi \in \text{set } \varphi$ s. progress σ P φ j)*
using *atms_nonempty_progress to_mregex_ok unfolding progress_regex_def by fastforce*

lemma *to_mregex_progress: safe_regex m g r \implies to_mregex r = (mr, φ s) \implies progress_regex σ P r j = (if φ s = [] then j else (MIN $\varphi \in \text{set } \varphi$ s. progress σ P φ j))*
using *to_mregex_nonempty_progress to_mregex_empty_progress unfolding progress_regex_def by auto*

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, 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 ($\lambda \varphi i. qtable n (fv \varphi) (mem_restr R) (\lambda v. Formula.sat \sigma V (map the v) i \varphi)) \varphi$ 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'*
shows *wf_mbufn' σ P' V j' n R r buf'*
and *wf_ts_regex σ P' j' r nts'*
using *pre_buf pre_nts bounded rm mr safe xss $\langle j \leq j' \rangle$ unfolding wf_mbufn'_def wf_ts_regex_def*
using *atms_empty_progress[of m g r] to_mregex_ok[OF mr]*
by (*auto 0 3 simp: list_rel_map to_mregex_empty_progress to_mregex_nonempty_progress Mini_def*
intro: progress_mono_gen[OF $\langle j \leq j' \rangle$ rm] list_rel_refl_strong progress_le_gen
dest: list_all2_lengthD elim!: mbufnt_take_eqD[OF eq wf_mbufn_add])

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' σ P V j n R φ ψ buf*
and *pre_nts: list_all2 ($\lambda i t. t = \tau \sigma i$) [min (progress σ P φ j) (progress σ P ψ j).. j'] nts*
and *xs: list_all2 ($\lambda i. qtable n (Formula.fv \varphi) (mem_restr R) (\lambda v. Formula.sat \sigma V (map the v) i \varphi))$*
[progress σ P φ j.. \langle progress σ P' φ j' \rangle] xs
and *ys: list_all2 ($\lambda i. qtable n (Formula.fv \psi) (mem_restr R) (\lambda v. Formula.sat \sigma V (map the v) i \psi))$*
[progress σ P ψ j.. \langle progress σ P' ψ j' \rangle] ys
and *j \leq j'*
and *base: U (min (progress σ P φ j) (progress σ P ψ j)) z*
and *step: $\bigwedge k X Y z. \text{min} (progress \sigma P \varphi j) (progress \sigma P \psi j) \leq k \implies$*
Suc k \leq progress σ P' φ j' \implies Suc k \leq progress σ P' ψ j' \implies
qtable n (Formula.fv φ) (mem_restr R) ($\lambda v. Formula.sat \sigma V (map the v) k \varphi$) X \implies
qtable n (Formula.fv ψ) (mem_restr R) ($\lambda v. Formula.sat \sigma V (map the v) k \psi$) Y \implies
U k z \implies U (Suc k) (f X Y ($\tau \sigma k$) z)
shows *U (min (progress σ P' φ j') (progress σ P' ψ j')) z'*
using *pre_buf pre_nts bounded rm unfolding wf_mbuf2'_def*
by (*blast intro!: mbuf2t_take_induct[OF eq] wf_mbuf2_add[OF _ xs ys] progress_mono_gen[OF $\langle j \leq j' \rangle$ rm]*
progress_le_gen base step)

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'* $\sigma P V j n R r$ *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* ($\lambda \varphi i. qtable\ n\ (fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi)$) 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* $\sigma P r j$) *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* $\sigma P' r j'$) *z'*
using *pre_buf pre_nts bounded rm* $\langle j \leq j' \rangle$ *to_mregex_progress*[*OF safe mr, of* $\sigma P' j'$] *to_mregex_empty_progress*[*OF*
safe, of $mr\ \sigma P j$] *base*
unfolding *wf_mbufn'_def mr prod.case*
by (*fastforce* *dest!*: *mbufnt_take_induct*[*OF eq wf_mbufn_add*[*OF* $_ xss$] *pre_nts*, *of* *U*]
simp: *list.rel_map le_imp_diff_is_add ac_simps Mini_def*
intro: *progress_mono_gen*[*OF* $\langle j \leq j' \rangle rm$] *list.rel_refl_strong progress_le_gen*
intro!: *base step dest: list_all2_lengthD split: if_splits*)

lemma *progress_Until_le*: *progress* $\sigma P (Formula.Until\ \varphi\ I\ \psi)$ $j \leq \min (progress\ \sigma P\ \varphi\ j) (progress\ \sigma P\ \psi\ j)$
by (*cases* *right* *I*) (*auto simp: trans_le_add1 intro!: cInf_lower*)

lemma *progress_MatchF_le*: *progress* $\sigma P (Formula.MatchF\ I\ r)$ $j \leq progress_regex\ \sigma P r j$
by (*cases* *right* *I*) (*auto simp: trans_le_add1 progress_regex_def intro!: cInf_lower*)

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*)
by (*simp* *add: list_all2_Cons2 upt_eq_Cons_conv*)

lemma *list_all2_upt_append*: *list_all2* *P* [*a..<b*] *xs* $\implies list_all2\ P\ [b..<c]\ ys \implies$
 $a \leq b \implies b \leq c \implies list_all2\ P\ [a..<c]\ (xs\ @\ ys)$
by (*induction* *xs* *arbitrary: a*) (*auto simp* *add: list_all2_Cons2 upt_eq_Cons_conv*)

lemma *list_all3_list_all2_conv*: *list_all3* *R xs xs ys = list_all2* ($\lambda x. R\ x\ x$) *xs ys*
by (*auto simp: list_all3_conv_all_nth list_all2_conv_all_nth*)

lemma *map_split_map*: *map_split* *f* (*map* *g xs*) = *map_split* (*f* *o* *g*) *xs*
by (*induct* *xs*) *auto*

lemma *map_split_alt*: *map_split* *f xs* = (*map* (*fst* *o* *f*) *xs*, *map* (*snd* *o* *f*) *xs*)
by (*induct* *xs*) (*auto split: prod.splits*)

lemma *fv_formula_of_constraint*: *fv* (*formula_of_constraint* (*t1*, *p*, *c*, *t2*)) = *fv_trm* *t1* \cup *fv_trm* *t2*
by (*induction* (*t1*, *p*, *c*, *t2*) *rule: formula_of_constraint.induct*) *simp_all*

lemma (*in* *maux*) *wf_mformula_wf_set*: *wf_mformula* $\sigma j P V n R \varphi \varphi' \implies wf_set\ n\ (Formula.fv\ \varphi')$
unfolding *wf_set_def*
proof (*induction* *rule: wf_mformula.induct*)
case (*AndRel* *P V n R $\varphi \varphi' \psi'$ conf*)
then show ?*case* **by** (*auto simp: fv_formula_of_constraint dest!: subsetD*)
next
case (*Ands* *P V n R l l_pos l_neg l' buf A_pos A_neg*)

```

from Ands.IH have  $\forall \varphi' \in \text{set } (l\_pos \ @ \ \text{map } \text{remove\_neg } l\_neg). \forall x \in \text{fv } \varphi'. x < n$ 
  by (simp add: list_all2_conv_all_nth all_set_conv_all_nth[of _ @ _] del: set_append)
then have  $\forall \varphi' \in \text{set } (l\_pos \ @ \ l\_neg). \forall x \in \text{fv } \varphi'. x < n$ 
  by (auto dest: bspec[where x=remove_neg _])
then show ?case using Ands.hyps(2) by auto
next
case (Agg P V b n R  $\varphi$   $\varphi'$  y f g0  $\omega$ )
then have Formula.fvi_trm b f  $\subseteq$  Formula.fvi b  $\varphi'$ 
  by (auto simp: fvi_trm_iff_fv_trm[where b=b] fvi_iff_fv[where b=b])
with Agg show ?case by (auto 0 3 simp: Un_absorb2 fvi_iff_fv[where b=b])
next
case (MatchP r P V n R  $\varphi$ s mr mrs buf nts I aux)
then obtain  $\varphi$ s' where conv: to_mregex r = (mr,  $\varphi$ s') by blast
with MatchP have  $\forall \varphi' \in \text{set } \varphi$ s'.  $\forall x \in \text{fv } \varphi'. x < n$ 
  by (simp add: list_all2_conv_all_nth all_set_conv_all_nth[of  $\varphi$ s'])
with conv show ?case
  by (simp add: to_mregex_ok[THEN conjunct1] fv_regex_alt[OF  $\langle$ safe_regex __  $\rangle$ ])
next
case (MatchF r P V n R  $\varphi$ s mr mrs buf nts I aux)
then obtain  $\varphi$ s' where conv: to_mregex r = (mr,  $\varphi$ s') by blast
with MatchF have  $\forall \varphi' \in \text{set } \varphi$ s'.  $\forall x \in \text{fv } \varphi'. x < n$ 
  by (simp add: list_all2_conv_all_nth all_set_conv_all_nth[of  $\varphi$ s'])
with conv show ?case
  by (simp add: to_mregex_ok[THEN conjunct1] fv_regex_alt[OF  $\langle$ safe_regex __  $\rangle$ ])
qed (auto simp: fvi_Suc split: if_splits)

```

lemma *qtable_mmulti_join*:

```

assumes pos: list_all3 ( $\lambda A \ Q_i \ X. \text{qtable } n \ A \ P \ Q_i \ X \wedge \text{wf\_set } n \ A$ ) A_pos Q_pos L_pos
and neg: list_all3 ( $\lambda A \ Q_i \ X. \text{qtable } n \ A \ P \ Q_i \ X \wedge \text{wf\_set } n \ A$ ) 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:  $\bigwedge x. \text{wf\_tuple } n \ C \ x \implies P \ x \implies \text{list\_all } (\lambda A. P (\text{restrict } A \ x)) \ A\_pos$ 
and restrict_neg:  $\bigwedge x. \text{wf\_tuple } n \ C \ x \implies P \ x \implies \text{list\_all } (\lambda A. P (\text{restrict } A \ x)) \ A\_neg$ 
and Qs:  $\bigwedge x. \text{wf\_tuple } n \ C \ x \implies P \ x \implies Q \ x \longleftrightarrow$ 
  list_all2 ( $\lambda A \ Q_i. Q_i (\text{restrict } A \ x)$ ) A_pos Q_pos  $\wedge$ 
  list_all2 ( $\lambda A \ Q_i. \neg Q_i (\text{restrict } A \ x)$ ) A_neg Q_neg
shows qtable n C P Q (mmulti_join n A_pos A_neg L)

```

proof (*rule qtableI*)

```

from pos have 1: list_all2 ( $\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A$ ) A_pos L_pos
  by (simp add: list_all3_conv_all_nth list_all2_conv_all_nth qtable_def)
moreover from neg have list_all2 ( $\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A$ ) A_neg L_neg
  by (simp add: list_all3_conv_all_nth list_all2_conv_all_nth qtable_def)
ultimately have L: list_all2 ( $\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A$ ) (A_pos @ A_neg) (L_pos @ L_neg)
  by (rule list_all2_appendI)
note in_join_iff = mmulti_join_correct[OF  $\langle$ A_pos  $\neq$  [] $\rangle$  L]
from 1 have take_eq: take (length A_pos) (L_pos @ L_neg) = L_pos
  by (auto dest!: list_all2_lengthD)
from 1 have drop_eq: drop (length A_pos) (L_pos @ L_neg) = L_neg
  by (auto dest!: list_all2_lengthD)

```

```

note mmulti_join.simps[simp del]
show table n C (mmulti_join n A_pos A_neg L)
  unfolding C_eq L_eq table_def by (simp add: in_join_iff)
show Q x if x  $\in$  mmulti_join n A_pos A_neg L wf_tuple n C x P x for x
using that(2,3)
proof (rule Qs[THEN iffD2, OF __ conjI])
  have pos': list_all2 ( $\lambda A. (\in) (\text{restrict } A \ x)$ ) A_pos L_pos
  and neg': list_all2 ( $\lambda A. (\notin) (\text{restrict } A \ x)$ ) A_neg L_neg

```

```

    using that(1) unfolding L_eq in_join_iff take_eq drop_eq by simp_all
  show list_all2 (λA Qi. Qi (restrict A x)) A_pos Q_pos
    using pos pos' restrict_pos that(2,3)
  by (simp add: list_all3_conv_all_nth list_all2_conv_all_nth list_all_length qtable_def)
  have fv_subset': λi. i < length A_neg ⇒ A_neg ! i ⊆ C
    using fv_subset unfolding C_eq by (auto simp: Sup_le_iff)
  show list_all2 (λA Qi. ¬ Qi (restrict A x)) A_neg Q_neg
    using neg neg' restrict_neg that(2,3)
  by (auto simp: list_all3_conv_all_nth list_all2_conv_all_nth list_all_length qtable_def
    wf_tuple_restrict_simple[OF fv_subset'])
qed
show x ∈ mmulti_join n A_pos A_neg L if wf_tuple n C x P x Q x for x
  unfolding L_eq in_join_iff take_eq drop_eq
proof (intro conjI)
  from that have pos': list_all2 (λA Qi. Qi (restrict A x)) A_pos Q_pos
  and neg': list_all2 (λA Qi. ¬ Qi (restrict A x)) A_neg Q_neg
  using Qs[THEN iffD1] by auto
  show wf_tuple n (⋃ A ∈ set A_pos. A) x
    using ⟨wf_tuple n C x⟩ unfolding C_eq by simp
  show list_all2 (λA. (∈) (restrict A x)) A_pos L_pos
    using pos pos' restrict_pos that(1,2)
  by (simp add: list_all3_conv_all_nth list_all2_conv_all_nth list_all_length qtable_def
    C_eq wf_tuple_restrict_simple[OF Sup_upper])
  show list_all2 (λA. (∉) (restrict A x)) A_neg L_neg
    using neg neg' restrict_neg that(1,2)
  by (auto simp: list_all3_conv_all_nth list_all2_conv_all_nth list_all_length qtable_def)
qed
qed

lemma nth_filter: i < length (filter P xs) ⇒
  (λi'. i' < length xs ⇒ P (xs ! i') ⇒ Q (xs ! i')) ⇒ Q (filter P xs ! i)
  by (metis (lifting) in_set_conv_nth set_filter mem_Collect_eq)

lemma nth_partition: i < length xs ⇒
  (λi'. i' < length (filter P xs) ⇒ Q (filter P xs ! i')) ⇒
  (λi'. i' < length (filter (Not ∘ P) xs) ⇒ Q (filter (Not ∘ P) xs ! i')) ⇒ Q (xs ! i)
  by (metis (no_types, lifting) in_set_conv_nth set_filter mem_Collect_eq comp_apply)

lemma qtable_bin_join:
  assumes qtable n A P Q1 X qtable n B P Q2 Y ¬ b ⇒ B ⊆ A C = A ∪ B
  λx. wf_tuple n C x ⇒ P x ⇒ P (restrict A x) ∧ P (restrict B x)
  λx. b ⇒ wf_tuple n C x ⇒ P x ⇒ Q x ⇔ Q1 (restrict A x) ∧ Q2 (restrict B x)
  λx. ¬ b ⇒ wf_tuple n C x ⇒ P x ⇒ Q x ⇔ Q1 (restrict A x) ∧ ¬ Q2 (restrict B x)
  shows qtable n C P Q (bin_join n A X b B Y)
  using qtable_join[OF assms] bin_join_table[of n A X B Y b] assms(1,2)
  by (auto simp add: qtable_def)

lemma restrict_update: y ∉ A ⇒ y < length x ⇒ restrict A (x[y:=z]) = restrict A x
  unfolding restrict_def by (auto simp add: nth_list_update)

lemma qtable_assign:
  assumes qtable n A P Q X
  y < n insert y A = A' y ∉ A
  λx'. wf_tuple n A' x' ⇒ P x' ⇒ P (restrict A x')
  λx. wf_tuple n A x ⇒ P x ⇒ Q x ⇒ Q' (x[y:=Some (f x)])
  λx'. wf_tuple n A' x' ⇒ P x' ⇒ Q' x' ⇒ Q (restrict A x') ∧ x' ! y = Some (f (restrict A x'))
  shows qtable n A' P Q' ((λx. x[y:=Some (f x)]) ' X) (is qtable ___ ?Y)
proof (rule qtableI)

```

```

from assms(1) have table n A X unfolding qtable_def by simp
then show table n A ?Y
  unfolding table_def wf_tuple_def using assms(2,3)
  by (auto simp: nth_list_update)
next
fix x'
assume  $x' \in ?Y$  wf_tuple n A' x' P x'
then obtain x where  $x \in X$  and  $x'_{eq}: x' = x[y:=Some (f x)]$  by blast
then have wf_tuple n A x
  using assms(1) unfolding qtable_def table_def by blast
then have  $y < length\ x$  using assms(2) by (simp add: wf_tuple_def)
with  $\langle wf\_tuple\ n\ A\ x \rangle$  have restrict A  $x' = x$ 
  unfolding  $x'_{eq}$  by (simp add: restrict_update[OF assms(4)] restrict_idle)
with  $\langle wf\_tuple\ n\ A'\ x' \rangle$   $\langle P\ x' \rangle$  have P x
  using assms(5) by blast
with  $\langle wf\_tuple\ n\ A\ x \rangle$   $\langle x \in X \rangle$  have Q x
  using assms(1) by (elim in_qtableE)
with  $\langle wf\_tuple\ n\ A\ x \rangle$   $\langle P\ x \rangle$  show  $Q'\ x'$ 
  unfolding  $x'_{eq}$  by (rule assms(6))
next
fix x'
assume wf_tuple n A' x' P x' Q' x'
then have wf_tuple n A (restrict A x')
  using assms(3) by (auto intro!: wf_tuple_restrict_simple)
moreover have P (restrict A x')
  using  $\langle wf\_tuple\ n\ A'\ x' \rangle$   $\langle P\ x' \rangle$  by (rule assms(5))
moreover have Q (restrict A x') and  $y: x' ! y = Some (f (restrict\ A\ x'))$ 
  using  $\langle wf\_tuple\ n\ A'\ x' \rangle$   $\langle P\ x' \rangle$   $\langle Q'\ x' \rangle$  by (auto dest!: assms(7))
ultimately have restrict A  $x' \in X$  by (intro in_qtableI[OF assms(1)])
moreover have  $x' = (restrict\ A\ x')[y:=Some (f (restrict\ A\ x'))]$ 
  using y assms(2,3)  $\langle wf\_tuple\ n\ A (restrict\ A\ x') \rangle$   $\langle wf\_tuple\ n\ A'\ x' \rangle$ 
  by (auto simp: list_eq_iff_nth_eq wf_tuple_def nth_list_update nth_restrict)
ultimately show  $x' \in ?Y$  by simp
qed

```

lemma *sat_the_update*: $y \notin fv\ \varphi \implies Formula.sat\ \sigma\ V (map\ the\ (x[y:=z]))\ i\ \varphi = Formula.sat\ \sigma\ V (map\ the\ x)\ i\ \varphi$
by (*rule sat_fv_cong*) (*metis map_update_nth_list_update_neq*)

lemma *progress_constraint*: *progress* σ *P* (*formula_of_constraint* *c*) $j = j$
by (*induction rule: formula_of_constraint.induct*) *simp_all*

lemma *qtable_filter*:
assumes *qtable* *n* *A* *P* *Q* *X*
 $\bigwedge x. wf_tuple\ n\ A\ x \implies P\ x \implies Q\ x \wedge R\ x \longleftrightarrow Q'\ x$
shows *qtable* *n* *A* *P* *Q'* (*Set.filter* *R* *X*) (**is** *qtable* $_ _ _ _ ?Y$)
proof (*rule qtableI*)
from *assms*(1) **have** *table* *n* *A* *X*
unfolding *qtable_def* **by** *simp*
then show *table* *n* *A* *?Y*
unfolding *table_def* *wf_tuple_def* **by** *simp*
next
fix *x*
assume $x \in ?Y$ *wf_tuple* *n* *A* *x* *P* *x*
with *assms* **show** $Q'\ x$ **by** (*auto elim!: in_qtableE*)
next
fix *x*
assume *wf_tuple* *n* *A* *x* *P* *x* *Q'* *x*

with *assms* **show** $x \in \text{Set.filter } R \ X$ **by** (*auto intro!*: *in_qtableI*)
qed

lemma *eval_constraint_sat_eq*: $\text{wf_tuple } n \ A \ x \implies \text{fv_trm } t1 \subseteq A \implies \text{fv_trm } t2 \subseteq A \implies$
 $\forall i \in A. i < n \implies \text{eval_constraint } (t1, p, c, t2) \ x =$
 $\text{Formula.sat } \sigma \ V \ (\text{map the } x) \ i \ (\text{formula_of_constraint } (t1, p, c, t2))$
by (*induction* (*t1*, *p*, *c*, *t2*) *rule*: *formula_of_constraint.induct*)
(*simp_all* *add*: *meval_trm_eval_trm*)

declare *progress_le_gen*[*simp*]

definition *wf_envs* $\sigma \ j \ P \ P' \ V \ db =$
(*dom* *V* = *dom* *P* \wedge
Mapping.keys *db* = *dom* *P* \cup $\{p. p \in \text{fst } \Gamma \ \sigma \ j\}$ \wedge
rel_mapping (\leq) *P* *P'* \wedge
pred_mapping ($\lambda i. i \leq j$) *P* \wedge
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* \times *event_data* *list*) *set* \Rightarrow *Formula.database* **is**
 $\lambda X \ p. (\text{if } p \in \text{fst } X \ \text{then } \text{Some } [\{ts. (p, ts) \in X\}] \ \text{else } \text{None}) .$

lemma *wf_envs_mk_db*: *wf_envs* $\sigma \ j \ \text{Map.empty } \ \text{Map.empty } \ \text{Map.empty} \ (\text{mk_db } (\Gamma \ \sigma \ j))$
unfolding *wf_envs_def* *mk_db_def*
by *transfer* (*force split*: *if_splits simp*: *image_iff rel_mapping_alt*)

lemma *wf_envs_update*:

assumes *wf_envs*: *wf_envs* $\sigma \ j \ P \ P' \ V \ db$
and *m_eq*: $m = \text{Formula.nfv } \varphi$
and *in_fv*: $\{0 ..< m\} \subseteq \text{fv } \varphi$
and *xs*: *list_all2* ($\lambda i. \text{qtable } m \ (\text{Formula.fv } \varphi) \ (\text{mem_restr } UNIV) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi)$)
 $[\text{progress } \sigma \ P \ \varphi \ j..<\text{progress } \sigma \ P' \ \varphi \ (\text{Suc } j)] \ xs$
shows *wf_envs* $\sigma \ j \ (P(p \mapsto \text{progress } \sigma \ P \ \varphi \ j)) \ (P'(p \mapsto \text{progress } \sigma \ P' \ \varphi \ (\text{Suc } j)))$
 $(V(p \mapsto \lambda i. \{v. \text{length } v = m \wedge \text{Formula.sat } \sigma \ V \ v \ i \ \varphi\}))$
 $(\text{Mapping.update } p \ (\text{map } (\text{image } (\text{map the})) \ xs) \ db)$
unfolding *wf_envs_def*
proof (*intro conjI ballI, goal_cases*)
case 3
from *assms* **show** ?*case*
by (*auto simp*: *wf_envs_def pred_mapping_alt progress_le progress_mono_gen*
intro!: *rel_mapping_map_upd*)
next
case (6 *p'*)
with *assms* **show** ?*case* **by** (*cases* $p' \in \text{dom } P$) (*auto simp*: *wf_envs_def lookup_update'*)
next
case (7 *p'*)
from *xs in_fv* **have** *list_all2* ($\lambda x \ y. \text{map the } 'y = \{v. \text{length } v = m \wedge \text{Formula.sat } \sigma \ V \ v \ x \ \varphi\}$)
 $[\text{progress } \sigma \ P \ \varphi \ j..<\text{progress } \sigma \ P' \ \varphi \ (\text{Suc } j)] \ xs$
by (*elim* *list.rel_mono_strong*) (*auto* 0 3 *simp*: *wf_tuple_def nth_append*
elim!: *in_qtableE in_qtableI intro!*: *image_eqI*[**where** $x = \text{map } \text{Some } _$])
moreover **have** *list_all2* ($\lambda i \ X. X = \text{the } (V \ p') \ i$) [*the* (*P p'*)..*the* (*P' p'*)] (*the* (*Mapping.lookup* *db*
p'))
if $p \neq p'$
proof –
from *that* 7 **have** $p' \in \text{dom } P$ **by** *simp*

```

    with wf_envs show ?thesis by (simp add: wf_envs_def)
qed
ultimately show ?case
  by (simp add: list.rel_map image_iff lookup_update')
qed (use assms in ‹auto simp: wf_envs_def›)

lemma wf_envs_P_simps[simp]:
  wf_envs σ j P P' V db ⇒ pred_mapping (λi. i ≤ j) P
  wf_envs σ j P P' V db ⇒ pred_mapping (λi. i ≤ Suc j) P'
  wf_envs σ j P P' V db ⇒ rel_mapping (≤) P P'
unfolding wf_envs_def by auto

lemma wf_envs_progress_le[simp]:
  wf_envs σ j P P' V db ⇒ progress σ P φ j ≤ j
  wf_envs σ j P P' V db ⇒ progress σ P' φ (Suc j) ≤ Suc j
unfolding wf_envs_def by auto

lemma wf_envs_progress_regeq_le[simp]:
  wf_envs σ j P P' V db ⇒ progress_regeq σ P r j ≤ j
  wf_envs σ j P P' V db ⇒ progress_regeq σ P' r (Suc j) ≤ Suc j
unfolding wf_envs_def by (auto simp: progress_regeq_le)

lemma wf_envs_progress_mono[simp]:
  wf_envs σ j P P' V db ⇒ a ≤ b ⇒ progress σ P φ a ≤ progress σ P' φ b
unfolding wf_envs_def
by (auto simp: progress_mono_gen)

lemma qtable_wf_tuple_cong: qtable n A P Q X ⇒ A = B ⇒ (∧v. wf_tuple n A v ⇒ P v ⇒ Q
v = Q' v) ⇒ qtable n B P Q' X
unfolding qtable_def table_def by blast

lemma (in maux) meval:
  assumes wf_mformula σ j P V n R φ φ' wf_envs σ j P P' V db
  shows case meval n (τ σ j) db φ of (xs, φ_n) ⇒ wf_mformula σ (Suc j) P' V n R φ_n φ' ∧
  list_all2 (λi. qtable n (Formula.fv φ') (mem_restr R) (λv. Formula.sat σ V (map the v) i φ'))
  [progress σ P φ' j..<progress σ P' φ' (Suc j)] xs
  using assms
proof (induction φ arbitrary: db P P' V n R φ')
  case (MRel rel)
  then show ?case
    by (cases rule: wf_mformula.cases)
      (auto simp add: ball_Un intro: wf_mformula.intros table_eq_rel eq_rel_eval_trm
in_eq_rel qtable_empty qtable_unit_table intro!: qtableI)
next
  case (MPred e ts)
  then show ?case
  proof (cases e ∈ dom P)
    case True
    with MPred(2) have e ∈ Mapping.keys db e ∈ dom P' e ∈ dom V
      list_all2 (λi X. X = the (V e) i) [the (P e)..<the (P' e)]
      (the (Mapping.lookup db e)) unfolding wf_envs_def rel_mapping_alt by blast+
    with MPred(1) True show ?thesis
      by (cases rule: wf_mformula.cases)
        (fastforce intro!: wf_mformula.Pred qtableI bexI[where P=λx. _ = tabulate x 0 n, OF refl]
elim!: list.rel_mono_strong bexI[rotated] dest: ex_match
simp: list.rel_map table_def match_wf_tuple in_these_eq match_eval_trm image_iff
list.map_comp keys_dom_lookup)
  next

```

```

note MPred(1)
moreover
case False
moreover
from False MPred(2) have  $e \notin \text{dom } P' \ e \notin \text{dom } V$ 
  unfolding wf_envs_def rel_mapping_alt by auto
moreover
from False MPred(2) have *:  $e \in \text{fst } \Gamma \ \sigma \ j \iff e \in \text{Mapping.keys } db$ 
  unfolding wf_envs_def by auto
from False MPred(2) have
   $e \in \text{Mapping.keys } db \implies \text{Mapping.lookup } db \ e = \text{Some } [\{ts. (e, ts) \in \Gamma \ \sigma \ j\}]$ 
  unfolding wf_envs_def keys_dom_lookup by (metis Diff_iff domD option.sel)
with * have (case Mapping.lookup db e of None  $\Rightarrow$   $[\{\}]$  | Some xs  $\Rightarrow$  xs) =  $[\{ts. (e, ts) \in \Gamma \ \sigma \ j\}]$ 
  by (cases e  $\in$  fst '  $\Gamma \ \sigma \ j$ ) (auto simp: image_iff keys_dom_lookup split: option.splits)
ultimately show ?thesis
  by (cases rule: wf_mformula.cases)
    (fastforce intro!: wf_mformula.Pred qtableI bezI[where  $P = \lambda x. \_ = \text{tabulate } x \ 0 \ n, \text{OF refl}$ ]
    elim!: list.rel_mono_strong bezI[rotated] dest: ex_match
    simp: list.rel_map table_def match_wf_tuple in_these_eq match_eval_trm image_iff list.map_comp)
qed
next
case (MLet p m  $\varphi_1 \ \varphi_2$ )
from MLet.prem(1) obtain  $\varphi_1' \ \varphi_2'$  where Let:  $\varphi' = \text{Formula.Let } p \ \varphi_1' \ \varphi_2'$  and
  1: wf_mformula  $\sigma \ j \ P \ V \ m \ \text{UNIV} \ \varphi_1 \ \varphi_1'$  and
  fv:  $m = \text{Formula.nfv } \varphi_1' \ \{0..<m\} \subseteq \text{fv } \varphi_1'$  and
  2: wf_mformula  $\sigma \ j \ (P(p \mapsto \text{progress } \sigma \ P \ \varphi_1' \ j))$ 
    ( $V(p \mapsto \lambda i. \{v. \text{length } v = m \wedge \text{Formula.sat } \sigma \ V \ v \ i \ \varphi_1'\})$ )
     $n \ R \ \varphi_2 \ \varphi_2'$ 
  by (cases rule: wf_mformula.cases) auto
obtain xs  $\varphi_{1\_new}$  where e1: meval m ( $\tau \ \sigma \ j$ ) db  $\varphi_1 = (xs, \varphi_{1\_new})$  and
  wf1: wf_mformula  $\sigma \ (\text{Suc } j) \ P' \ V \ m \ \text{UNIV} \ \varphi_{1\_new} \ \varphi_1'$  and
  res1: list_all2 ( $\lambda i. \text{qtable } m \ (\text{fv } \varphi_1') \ (\text{mem\_restr } \text{UNIV}) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi_1')$ )
    [ $\text{progress } \sigma \ P \ \varphi_1' \ j..<\text{progress } \sigma \ P' \ \varphi_1' \ (\text{Suc } j)$ ] xs
  using MLet(1)[OF 1(1) MLet.prem(2)] by (auto simp: eqTrueI[OF mem_restr_UNIV, abs_def])
from MLet(2)[OF 2 wf_envs_update[OF MLet.prem(2) fv res1]] wf1 e1 fv
show ?case unfolding Let
  by (auto simp: fun_upd_def intro!: wf_mformula.Let)
next
case (MAnd A_ $\varphi \ \varphi$  pos A_ $\psi \ \psi$  buf)
from MAnd.prem show ?case
  by (cases rule: wf_mformula.cases)
    (auto simp: sat_the_restrict simp del: bin_join.simps
    dest!: MAnd.IH split: if_splits prod.splits intro!: wf_mformula.And qtable_bin_join
    elim: mbuf2_take_add'(1) list.rel_mono_strong[OF mbuf2_take_add'(2)])
next
case (MAndAssign  $\varphi$  conf)
from MAndAssign.prem obtain  $\varphi'' \ x \ t \ \psi''$  where
  wf_envs: wf_envs  $\sigma \ j \ P \ P' \ V \ db$  and
   $\varphi''_{eq}$ :  $\varphi' = \text{formula.And } \varphi'' \ \psi''$  and
  wf_ $\varphi$ : wf_mformula  $\sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi''$  and
   $x < n$  and
   $x \notin \text{fv } \varphi''$  and
  fv_t_subset: fv_trm  $t \subseteq \text{fv } \varphi''$  and
  conf:  $(x, t) = \text{conf}$  and
   $\psi''_{eqs}$ :  $\psi'' = \text{formula.Eq } (\text{trm.Var } x) \ t \vee \psi'' = \text{formula.Eq } t \ (\text{trm.Var } x)$ 
  by (cases rule: wf_mformula.cases)
from wf_ $\varphi$  wf_envs obtain xs  $\varphi_n$  where
  meval_eq: meval n ( $\tau \ \sigma \ j$ ) db  $\varphi = (xs, \varphi_n)$  and

```

```

wf_φn: wf_mformula σ (Suc j) P' V n R φn φ'' and
xs: list_all2 (λi. qtable n (fv φ'') (mem_restr R) (λv. Formula.sat σ V (map the v) i φ'))
  [progress σ P φ' j..<progress σ P' φ'' (Suc j)] xs
by (auto dest!: MAndAssign.IH)
have progress_eqs:
  progress σ P φ' j = progress σ P φ'' j
  progress σ P' φ' (Suc j) = progress σ P' φ'' (Suc j)
using ψ''_eqs wf_envs_progress_le[OF wf_envs] by (auto simp: φ'_eq)

show ?case proof (simp add: meval_eq, intro conjI)
  show wf_mformula σ (Suc j) P' V n R (MAndAssign φn conf) φ'
    unfolding φ'_eq
    by (rule wf_mformula.AndAssign) fact+
next
show list_all2 (λi. qtable n (fv φ') (mem_restr R) (λv. Formula.sat σ V (map the v) i φ'))
  [progress σ P φ' j..<progress σ P' φ' (Suc j)] (map ((·) (eval_assignment conf)) xs)
  unfolding list.rel_map progress_eqs conf[symmetric] eval_assignment.simps
  using xs
proof (rule list.rel_mono_strong)
  fix i X
  assume qtable: qtable n (fv φ'') (mem_restr R) (λv. Formula.sat σ V (map the v) i φ'') X
  then show qtable n (fv φ') (mem_restr R) (λv. Formula.sat σ V (map the v) i φ')
    ((λy. y[x := Some (meval_trm t y)]) ' X)
  proof (rule qtable_assign)
    show x < n by fact
    show insert x (fv φ'') = fv φ'
      using ψ''_eqs fv_t_subset by (auto simp: φ'_eq)
    show x ∉ fv φ'' by fact
  next
  fix v
  assume wf_v: wf_tuple n (fv φ') v and mem_restr R v
  then show mem_restr R (restrict (fv φ'') v) by simp

  assume sat: Formula.sat σ V (map the v) i φ'
  then have A: Formula.sat σ V (map the (restrict (fv φ'') v)) i φ'' (is ?A)
    by (simp add: φ'_eq sat_the_restrict)
  have map_the v ! x = Formula.eval_trm (map the v) t
    using sat ψ''_eqs by (auto simp: φ'_eq)
  also have ... = Formula.eval_trm (map the (restrict (fv φ'') v)) t
    using fv_t_subset by (auto simp: map_the_restrict intro!: eval_trm_fv_cong)
  finally have map_the v ! x = meval_trm t (restrict (fv φ'') v)
    using meval_trm_eval_trm[of n fv φ'' restrict (fv φ'') v t]
    fv_t_subset wf_v wf_mformula_wf_set[unfolded wf_set_def, OF wf_φ]
    by (fastforce simp: φ'_eq intro!: wf_tuple_restrict)
  then have B: v ! x = Some (meval_trm t (restrict (fv φ'') v)) (is ?B)
    using ψ''_eqs wf_v ⟨x < n⟩ by (auto simp: wf_tuple_def φ'_eq)
  from A B show ?A ∧ ?B ..
next
  fix v
  assume wf_v: wf_tuple n (fv φ'') v and mem_restr R v
  and sat: Formula.sat σ V (map the v) i φ''
  let ?v = v[x := Some (meval_trm t v)]
  from sat have A: Formula.sat σ V (map the ?v) i φ''
    using ⟨x ∉ fv φ''⟩ by (simp add: sat_the_update)
  have y ∈ fv_trm t ⇒ x ≠ y for y
    using fv_t_subset ⟨x ∉ fv φ''⟩ by auto
  then have B: Formula.sat σ V (map the ?v) i ψ''
    using ψ''_eqs meval_trm_eval_trm[of n fv φ'' v t] ⟨x < n⟩

```

```

      fv_t_subset wf_v wf_mformula_wf_set[unfolded wf_set_def, OF wf_φ]
    by (auto simp: wf_tuple_def map_update intro!: eval_trm_fv_cong)
  from A B show Formula.sat σ V (map the ?v) i φ'
    by (simp add: φ'_eq)
  qed
  qed
  qed
next
case (MAndRel φ conf)
from MAndRel.premis show ?case
  by (cases rule: wf_mformula.cases)
    (auto simp: progress_constraint progress_le list_rel_map fv_formula_of_constraint
      Un_absorb2 wf_mformula_wf_set[unfolded wf_set_def] simp del: Set.filter_eq split: prod.splits
      dest!: MAndRel.IH[where db=db and P=P and P'=P'] eval_constraint_sat_eq[THEN iffD2]
      intro!: wf_mformula.AndRel
      elim!: list_rel_mono_strong qtable_filter eval_constraint_sat_eq[THEN iffD1])
next
case (MAnds A_pos A_neg l buf)
note mbufn_take.simps[simp del] mbufn_add.simps[simp del] mmulti_join.simps[simp del]

from MAnds.premis obtain pos neg l' where
  wf_l: list_all2 (wf_mformula σ j P V n R) l (pos @ map remove_neg neg) and
  wf_buf: wf_mbufn (progress σ P (formula.Ands l') j) (map (λψ. progress σ P ψ j) (pos @ map
remove_neg neg))
  (map (λψ i. qtable n (fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) i ψ)) (pos @ map
remove_neg neg)) buf and
  posneg: (pos, neg) = partition safe_formula l' and
  pos ≠ [] and
  safe_neg: list_all safe_formula (map remove_neg neg) and
  A_eq: A_pos = map fv pos A_neg = map fv neg and
  fv_subset: ⋃ (set A_neg) ⊆ ⋃ (set A_pos) and
  φ' = Formula.Ands l'
  by (cases rule: wf_mformula.cases) simp
have progress_eq: progress σ P' (formula.Ands l') (Suc j) =
  Mini (progress σ P (formula.Ands l') j) (map (λψ. progress σ P' ψ (Suc j)) (pos @ map remove_neg
neg))
  using ⟨pos ≠ []⟩ posneg
  by (auto simp: Mini_def image_Un[symmetric] Collect_disj_eq[symmetric] intro!: arg_cong[where
f=Min])

have join_ok: qtable n (⋃ (fv ' set l')) (mem_restr R)
  (λv. list_all (Formula.sat σ V (map the v) k) l')
  (mmulti_join n A_pos A_neg L)
  if args_ok: list_all2 (λx. qtable n (fv x) (mem_restr R) (λv. Formula.sat σ V (map the v) k x))
  (pos @ map remove_neg neg) L
  for k L
proof (rule qtable_mmulti_join)
  let ?ok = λA Qi X. qtable n A (mem_restr R) Qi X ∧ wf_set n A
  let ?L_pos = take (length A_pos) L
  let ?L_neg = drop (length A_pos) L
  have 1: length pos ≤ length L
    using args_ok by (auto dest!: list_all2_lengthD)
  show list_all3 ?ok A_pos (map (λψ v. Formula.sat σ V (map the v) k ψ) pos) ?L_pos
    using args_ok wf_l unfolding A_eq
    by (auto simp add: list_all3_conv_all_nth list_all2_conv_all_nth nth_append
      split: if_splits intro!: wf_mformula_wf_set[of σ j P V n R]
      dest: order.strict_trans2[OF _ 1])
  from args_ok have prems_neg: list_all2 (λψ. qtable n (fv ψ) (mem_restr R) (λv. Formula.sat σ V

```

```

(map the v) k (remove_neg ψ)) neg ?L_neg
  by (auto simp: A_eq list_all2_append1 list.rel_map)
  show list_all3 ?ok A_neg (map (λψ v. Formula.sat σ V (map the v) k (remove_neg ψ)) neg) ?L_neg
  using prems_neg wf_l unfolding A_eq
  by (auto simp add: list_all3_conv_all_nth list_all2_conv_all_nth list_all_length_nth_append
less_diff_conv
  split: if_splits intro!: wf_mformula_wf_set[of σ j P V n R _ remove_neg _, simplified])
  show  $\bigcup (fv \text{ ' set } l') = \bigcup (set A\_pos)$ 
  using fv_subset posneg unfolding A_eq by auto
  show L = take (length A_pos) L @ drop (length A_pos) L by simp
  show A_pos ≠ [] using ⟨pos ≠ []⟩ A_eq by simp

fix x :: event_data tuple
assume wf_tuple n ( $\bigcup (fv \text{ ' set } l')$ ) x and mem_restr R x
then show list_all (λA. mem_restr R (restrict A x)) A_pos
  and list_all (λA. mem_restr R (restrict A x)) A_neg
  by (simp_all add: list.pred_set)

have list_all Formula.is_Neg neg
  using posneg safe_neg
  by (auto 0 3 simp add: list.pred_map elim!: list.pred_mono_strong
  intro: formula.exhaust[of ψ Formula.is_Neg ψ for ψ])
then have list_all (λψ. Formula.sat σ V (map the v) i (remove_neg ψ)  $\longleftrightarrow$ 
 $\neg$  Formula.sat σ V (map the v) i ψ) neg for v i
  by (fastforce simp: Formula.is_Neg_def elim!: list.pred_mono_strong)
then show list_all (Formula.sat σ V (map the x) k) l' =
  (list_all2 (λA Qi. Qi (restrict A x)) A_pos
  (map (λψ v. Formula.sat σ V (map the v) k ψ) pos)  $\wedge$ 
  list_all2 (λA Qi.  $\neg$  Qi (restrict A x)) A_neg
  (map (λψ v. Formula.sat σ V (map the v) k
  (remove_neg ψ))
  neg))
  using posneg
  by (auto simp add: A_eq list_all2_conv_all_nth list_all_length sat_the_restrict
  elim: nth_filter nth_partition[where P=safe_formula and Q=Formula.sat _ _ _])
qed fact

from MAnds.premis(2) show ?case
  unfolding ⟨φ' = Formula.Ands l'⟩
  by (auto 0 3 simp add: list.rel_map progress_eq map2_map_map list_all3_map
  list_all3_list_all2_conv list.pred_map
  simp del: set_append map_append progress_simps split: prod.splits
  intro!: wf_mformula.Ands[OF _ _ posneg ⟨pos ≠ []⟩ safe_neg A_eq fv_subset]
  list.rel_mono_strong[OF wf_l] wf_mbufn_add[OF wf_buf]
  list.rel_flip[THEN iffD1, OF list.rel_mono_strong, OF wf_l]
  list.rel_refl_join_ok[unfolded list.pred_set]
  dest!: MAnds.IH[OF _ _ MAnds.premis(2), rotated]
  elim!: wf_mbufn_take list_all2_appendI
  elim: mbufn_take_induct[OF _ wf_mbufn_add[OF wf_buf]])
next
case (MOr φ ψ buf)
from MOr.premis show ?case
  by (cases rule: wf_mformula.cases)
  (auto dest!: MOr.IH split: if_splits prod.splits intro!: wf_mformula.Or qtable_union
  elim: mbuf2_take_add'(1) list.rel_mono_strong[OF mbuf2_take_add'(2)])
next
case (MNeg φ)
have *: qtable n {} (mem_restr R) (λv. P v) X  $\implies$ 

```

```

    ¬ qtable n {} (mem_restr R) (λv. ¬ P v) empty_table ⇒ x ∈ X ⇒ False for P x X
  using nullary_qtable_cases qtable_unit_empty_table by fastforce
  from MNeg.premis show ?case
  by (cases rule: wf_mformula.cases)
    (auto 0 4 intro!: wf_mformula.Neg dest!: MNeg.IH
      simp add: list.rel_map
      dest: nullary_qtable_cases qtable_unit_empty_table intro!: qtable_empty_unit_table
      elim!: list.rel_mono_strong elim: *)
  next
  case (MExists φ)
  from MExists.premis show ?case
  by (cases rule: wf_mformula.cases)
    (force simp: list.rel_map fvi_Suc sat_fv_cong nth_Cons'
      intro!: wf_mformula.Exists dest!: MExists.IH qtable_project_fv
      elim!: list.rel_mono_strong table_fvi_tl qtable_cong sat_fv_cong[THEN iffD1, rotated -1]
      split: if_splits)+
  next
  case (MAgg g0 y ω b f φ)
  from MAgg.premis show ?case
  using wf_mformula_wf_set[OF MAgg.premis(1), unfolded wf_set_def]
  by (cases rule: wf_mformula.cases)
    (auto 0 3 simp add: list.rel_map simp del: sat.simps fvi.simps split: prod.split
      intro!: wf_mformula.Agg qtable_eval_agg dest!: MAgg.IH elim!: list.rel_mono_strong)
  next
  case (MPrev I φ first buf nts)
  from MPrev.premis show ?case
  proof (cases rule: wf_mformula.cases)
  case (Prev ψ)
  let ?xs = fst (meval n (τ σ j) db φ)
  let ?φ = snd (meval n (τ σ j) db φ)
  let ?ls = fst (mprev_next I (buf @ ?xs) (nts @ [τ σ j]))
  let ?rs = fst (snd (mprev_next I (buf @ ?xs) (nts @ [τ σ j])))
  let ?ts = snd (mprev_next I (buf @ ?xs) (nts @ [τ σ j]))
  let ?P = λi X. qtable n (fv ψ) (mem_restr R) (λv. Formula.sat σ V (map the v) i ψ) X
  let ?min = min (progress σ P' ψ (Suc j)) (Suc j - 1)
  from Prev MPrev.IH[OF _ MPrev.premis(2), of n R ψ] have IH: wf_mformula σ (Suc j) P' V n R
  ?φ ψ and
    list_all2 ?P [progress σ P ψ j..

```

```

min (progress  $\sigma$  P  $\psi$  j - Suc 0) (j - Suc 0) = progress  $\sigma$  P  $\psi$  j - Suc 0
min (progress  $\sigma$  P'  $\psi$  (Suc j) - Suc 0) j = progress  $\sigma$  P'  $\psi$  (Suc j) - Suc 0
using wf_envs_progress_le[OF MNext.premis(2), of  $\psi$ ] by auto

let ?xs = fst (meval n ( $\tau$   $\sigma$  j) db  $\varphi$ )
let ?ys = case (?xs, first) of (_ # xs, True)  $\Rightarrow$  xs | _  $\Rightarrow$  ?xs
let ? $\varphi$  = snd (meval n ( $\tau$   $\sigma$  j) db  $\varphi$ )
let ?ls = fst (mprev_next I ?ys (nts @ [ $\tau$   $\sigma$  j]))
let ?rs = fst (snd (mprev_next I ?ys (nts @ [ $\tau$   $\sigma$  j])))
let ?ts = snd (snd (mprev_next I ?ys (nts @ [ $\tau$   $\sigma$  j])))
let ?P =  $\lambda$  i X. qtable n (fv  $\psi$ ) (mem_restr R) ( $\lambda$  v. Formula.sat  $\sigma$  V (map the v) i  $\psi$ ) X
let ?min = min (progress  $\sigma$  P'  $\psi$  (Suc j) - 1) (Suc j - 1)
from Next MNext.IH[OF _ MNext.premis(2), of n R  $\psi$ ] have IH: wf_mformula  $\sigma$  (Suc j) P' V n R
? $\varphi$   $\psi$ 
  list_all2 ?P [progress  $\sigma$  P  $\psi$  j.. $\leq$ progress  $\sigma$  P'  $\psi$  (Suc j)] ?xs by auto
  with Next have list_all2 ( $\lambda$  i X. if mem ( $\tau$   $\sigma$  (Suc i) -  $\tau$   $\sigma$  i) I then ?P (Suc i) X else X =
empty_table)
    [progress  $\sigma$  P  $\psi$  j - 1.. $\leq$ ?min] ?ls  $\wedge$ 
    list_all2 ?P [Suc ?min.. $\leq$ progress  $\sigma$  P'  $\psi$  (Suc j)] ?rs  $\wedge$ 
    list_all2 ( $\lambda$  i t. t =  $\tau$   $\sigma$  i) [?min.. $\leq$ Suc j] ?ts if progress  $\sigma$  P  $\psi$  j  $\leq$  progress  $\sigma$  P'  $\psi$  (Suc j)
  using that wf_envs_progress_le[OF MNext.premis(2), of  $\psi$ ]
  by (intro mnext) (auto simp: list_all2_Cons2 upt_eq_Cons_conv
    intro!: list_all2_upt_append list_all2_appendI split: list.splits)
  then show ?thesis using wf_envs_progress_le[OF MNext.premis(2), of  $\psi$ ]
    wf_envs_progress_mono[OF MNext.premis(2), of j Suc j  $\psi$ , simplified] Next IH
  by (cases progress  $\sigma$  P'  $\psi$  (Suc j)  $>$  progress  $\sigma$  P  $\psi$  j)
    (auto 0 3 simp: qtable_empty_iff le_Suc_eq le_diff_conv
      elim!: wf_mformula.Next list.rel_mono_strong list_all2_appendI
      split: prod.split list.splits if_split_asm)

qed
next
case (MSince args  $\varphi$   $\psi$  buf nts aux)
note sat.simps[simp del]
from MSince.premis obtain  $\varphi''$   $\varphi'''$   $\psi''$  I where Since_eq:  $\varphi' =$  Formula.Since  $\varphi'''$  I  $\psi''$ 
and pos: if args_pos args then  $\varphi''' = \varphi''$  else  $\varphi''' =$  Formula.Neg  $\varphi''$ 
and pos_eq: safe_formula  $\varphi''' =$  args_pos args
and  $\varphi$ : wf_mformula  $\sigma$  j P V n R  $\varphi$   $\varphi''$ 
and  $\psi$ : wf_mformula  $\sigma$  j P V n R  $\psi$   $\psi''$ 
and fvi_subset: Formula.fv  $\varphi'' \subseteq$  Formula.fv  $\psi''$ 
and buf: wf_mbuf2'  $\sigma$  P V j n R  $\varphi''$   $\psi''$  buf
and nts: wf_ts  $\sigma$  P j  $\varphi''$   $\psi''$  nts
and aux: wf_since_aux  $\sigma$  V R args  $\varphi''$   $\psi''$  aux (progress  $\sigma$  P (Formula.Since  $\varphi'''$  I  $\psi''$ ) j)
and args_ivl: args_ivl args = I
and args_n: args_n args = n
and args_L: args_L args = Formula.fv  $\varphi''$ 
and args_R: args_R args = Formula.fv  $\psi''$ 
by (cases rule: wf_mformula.cases) (auto)
have  $\varphi'''$ : Formula.fv  $\varphi''' =$  Formula.fv  $\varphi''$  progress  $\sigma$  P  $\varphi'''$  j = progress  $\sigma$  P  $\varphi''$  j
progress  $\sigma$  P'  $\varphi'''$  j = progress  $\sigma$  P'  $\varphi''$  j for j
using pos by (simp_all split: if_splits)
from MSince.premis(2) have nts_snoc: list_all2 ( $\lambda$  i t. t =  $\tau$   $\sigma$  i)
[ $\min$  (progress  $\sigma$  P  $\varphi''$  j) (progress  $\sigma$  P  $\psi''$  j).. $\leq$ Suc j] (nts @ [ $\tau$   $\sigma$  j])
using nts unfolding wf_ts_def
by (auto simp add: wf_envs_progress_le[THEN min.coboundedI1] intro: list_all2_appendI)
have update: wf_since_aux  $\sigma$  V R args  $\varphi''$   $\psi''$  (snd (zs, aux')) (progress  $\sigma$  P' (Formula.Since  $\varphi'''$  I
 $\psi''$ ) (Suc j))  $\wedge$ 
list_all2 ( $\lambda$  i. qtable n (Formula.fv  $\varphi''' \cup$  Formula.fv  $\psi''$ ) (mem_restr R)
( $\lambda$  v. Formula.sat  $\sigma$  V (map the v) i (Formula.Since  $\varphi'''$  I  $\psi''$ )))

```

```

    [progress  $\sigma$  P (Formula.Since  $\varphi'''$  I  $\psi''$ ) j..<progress  $\sigma$  P' (Formula.Since  $\varphi'''$  I  $\psi''$ ) (Suc j)] (fst
(zs, aux'))
  if eval_φ: fst (meval n (τ σ j) db φ) = xs
  and eval_ψ: fst (meval n (τ σ j) db ψ) = ys
  and eq: mbuf2t_take (λr1 r2 t (zs, aux)).
    case update_since args r1 r2 t aux of (z, x) ⇒ (zs @ [z], x)
    ([], aux) (mbuf2t_add xs ys buf) (nts @ [τ σ j]) = ((zs, aux'), buf', nts')
  for xs ys zs aux' buf' nts'
  unfolding progress_simps φ'''
proof (rule mbuf2t_take_add_induct'[where j=j and j'=Suc j and z'=(zs, aux'),
  OF eq wf_envs_P_simps[OF MSince.prem(2)] buf nts_snoc],
  goal_cases xs ys _ base step)
  case xs
  then show ?case
  using MSince.IH(1)[OF φ MSince.prem(2)] eval_φ by auto
next
  case ys
  then show ?case
  using MSince.IH(2)[OF ψ MSince.prem(2)] eval_ψ by auto
next
  case base
  then show ?case
  using aux by (simp add: φ''')
next
  case (step k X Y z)
  then show ?case
  using fvi_subset pos
  by (auto 0 3 simp: args_ivl args_n args_L args_R Un_absorb1
    elim!: update_since(1) list_all2_appendI dest!: update_since(2)
    split: prod.split if_splits)
qed simp
with MSince.IH(1)[OF φ MSince.prem(2)] MSince.IH(2)[OF ψ MSince.prem(2)] show ?case
  by (auto 0 3 simp: Since_eq split: prod.split
    intro: wf_mformula.Since[OF _ _ pos pos_eq args_ivl args_n args_L args_R fvi_subset]
    elim: mbuf2t_take_add'(1)[OF _ wf_envs_P_simps[OF MSince.prem(2)] buf nts_snoc]
    mbuf2t_take_add'(2)[OF _ wf_envs_P_simps[OF MSince.prem(2)] buf nts_snoc])
next
  case (MUntil args φ ψ buf nts aux)
  note sat_simps[simp del] progress_simps[simp del]
  from MUntil.prem obtain φ'' φ''' ψ'' I where Until_eq: φ' = Formula.Until φ''' I ψ''
  and pos: if args_pos args then φ''' = φ'' else φ''' = Formula.Neg φ''
  and pos_eq: safe_formula φ''' = args_pos args
  and φ: wf_mformula σ j P V n R φ φ''
  and ψ: wf_mformula σ j P V n R ψ ψ''
  and fvi_subset: Formula.fv φ'' ⊆ Formula.fv ψ''
  and buf: wf_mbuf2t' σ P V j n R φ'' ψ'' buf
  and nts: wf_ts σ P j φ'' ψ'' nts
  and aux: wf_until_aux σ V R args φ'' ψ'' aux (progress σ P (Formula.Until φ''' I ψ'' j))
  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 length_aux: progress σ P (Formula.Until φ''' I ψ'' j) + length_muaux args aux =
    min (progress σ P φ'' j) (progress σ P ψ'' j)
  by (cases rule: wf_mformula.cases) (auto)
define pos where args_pos: pos = args_pos args
have φ''': progress σ P φ''' j = progress σ P φ'' j progress σ P φ''' j = progress σ P φ'' j for j
  using pos by (simp_all add: progress_simps split: if_splits)

```

```

from MUntil.prems(2) have nts_snoc: list_all2 ( $\lambda i t. t = \tau \sigma i$ )
  [min (progress  $\sigma P \varphi'' j$ ) (progress  $\sigma P \psi'' j$ )..Suc j] (nts @ [ $\tau \sigma j$ ])
using nts_unfolding wf_ts_def
by (auto simp add: wf_envs_progress_le[THEN min.coboundedI1] intro: list_all2_appendI)
{
  fix xs ys zs aux' aux'' buf' nts'
  assume eval_φ: fst (meval n ( $\tau \sigma j$ ) db  $\varphi$ ) = xs
    and eval_ψ: fst (meval n ( $\tau \sigma j$ ) db  $\psi$ ) = ys
    and eq1: mbuf2t_take (add_new_muaux args) aux (mbuf2_add xs ys buf) (nts @ [ $\tau \sigma j$ ]) =
      (aux', buf', nts')
    and eq2: eval_muaux args (case nts' of []  $\Rightarrow \tau \sigma j$  | nt # _  $\Rightarrow nt$ ) aux' = (zs, aux'')
  define ne where ne  $\equiv$  progress  $\sigma P$  (Formula.Until  $\varphi''' I \psi''$ ) j
  have update1: wf_until_aux  $\sigma V R$  args  $\varphi'' \psi'' aux'$  (progress  $\sigma P$  (Formula.Until  $\varphi''' I \psi''$ ) j)  $\wedge$ 
    ne + length_muaux args aux' = min (progress  $\sigma P' \varphi'''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j))
  using MUntil.IH(1)[OF  $\varphi$  MUntil.prems(2)] eval_φ MUntil.IH(2)[OF  $\psi$  MUntil.prems(2)]
    eval_ψ nts_snoc nts_snoc length_aux aux fvi_subset
  unfolding  $\varphi'''$ 
  by (elim mbuf2t_take_add_induct'[where j' = Suc j, OF eq1 wf_envs_P_simps[OF MUntil.prems(2)]]
    buf])
    (auto simp: args_n args_L args_R ne_def wf_update_until)
  then obtain cur auxlist' where valid_aux': valid_muaux args cur aux' auxlist' and
    cur: cur = (if ne + length_auxlist' = 0 then 0 else  $\tau \sigma$  (ne + length_auxlist' - 1)) and
    wf_auxlist': wf_until_auxlist  $\sigma V n R$  pos  $\varphi'' I \psi''$  auxlist' (progress  $\sigma P$  (Formula.Until  $\varphi''' I \psi''$ )
j)
  unfolding wf_until_aux_def ne_def args_ivl args_n args_pos by auto
  have length_aux': length_muaux args aux' = length_auxlist'
  using valid_length_muaux[OF valid_aux'] .
  have nts': wf_ts  $\sigma P'$  (Suc j)  $\varphi'' \psi'' nts'$ 
  using MUntil.IH(1)[OF  $\varphi$  MUntil.prems(2)] eval_φ MUntil.IH(2)[OF  $\psi$  MUntil.prems(2)]
    MUntil.prems(2) eval_ψ nts_snoc
  unfolding wf_ts_def
  by (intro mbuf2t_take_eqD(2)[OF eq1]) (auto intro: wf_mbuf2_add buf[unfolded wf_mbuf2'_def])
  define zs'' where zs'' = fst (eval_until I (case nts' of []  $\Rightarrow \tau \sigma j$  | nt # x  $\Rightarrow nt$ ) auxlist')
  define auxlist'' where auxlist'' = snd (eval_until I (case nts' of []  $\Rightarrow \tau \sigma j$  | nt # x  $\Rightarrow nt$ ) auxlist')
  have current_w_le: cur  $\leq$  (case nts' of []  $\Rightarrow \tau \sigma j$  | nt # x  $\Rightarrow nt$ )
  proof (cases nts')
  case Nil
  have p_le: min (progress  $\sigma P' \varphi'''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j)) - 1  $\leq j$ 
  using wf_envs_progress_le[OF MUntil.prems(2)]
  by (auto simp: min_def le_diff_conv)
  then show ?thesis
  unfolding cur conjunct2[OF update1, unfolded length_aux']
  using Nil by auto
  next
  case (Cons nt x)
  have progress_φ''': progress  $\sigma P' \varphi''$  (Suc j) = progress  $\sigma P' \varphi'''$  (Suc j)
  using pos by (auto simp add: progress_simps split: if_splits)
  have nt =  $\tau \sigma$  (min (progress  $\sigma P' \varphi''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j)))
  using nts'[unfolded wf_ts_def Cons]
  unfolding list_all2 Cons2 upt_eq Cons_conv by auto
  then show ?thesis
  unfolding cur conjunct2[OF update1, unfolded length_aux'] Cons progress_φ'''
  by (auto split: if_splits list_splits intro!:  $\tau$ _mono)
  qed
  have valid_aux'': valid_muaux args cur aux'' auxlist''
  using valid_eval_muaux[OF valid_aux' current_w_le eq2, of zs'' auxlist'']
  by (auto simp add: args_ivl zs''_def auxlist''_def)
  have length_aux'': length_muaux args aux'' = length_auxlist''

```

```

using valid_length_muaux[OF valid_aux'].
have eq2': eval_until I (case nts' of [] ⇒ τ σ j | nt # _ ⇒ nt) auxlist' = (zs, auxlist'')
using valid_eval_muaux[OF valid_aux' current_w_le eq2, of zs'' auxlist'']
by (auto simp add: args_ivl zs''_def auxlist''_def)
have length_aux'_aux'': length_muaux args aux' = length zs + length_muaux args aux''
using eval_until_length[OF eq2'] unfolding length_aux' length_aux''.
have i ≤ progress σ P' (Formula.Until φ''' I ψ'') (Suc j) ⇒
wf_until_auxlist σ V n R pos φ'' I ψ'' auxlist' i ⇒
i + length auxlist' = min (progress σ P' φ''' (Suc j)) (progress σ P' ψ'' (Suc j)) ⇒
wf_until_auxlist σ V n R pos φ'' I ψ'' auxlist'' (progress σ P' (Formula.Until φ''' I ψ'') (Suc j)) ∧
i + length zs = progress σ P' (Formula.Until φ''' I ψ'') (Suc j) ∧
i + length zs + length auxlist'' = min (progress σ P' φ''' (Suc j)) (progress σ P' ψ'' (Suc j)) ∧
list_all2 (λi. qtable n (Formula.fv ψ'') (mem_restr R)
(λv. Formula.sat σ V (map the v) i (Formula.Until (if pos then φ'' else Formula.Neg φ'') I ψ'')))
[i..<i + length zs] zs for i
using eq2'
proof (induction auxlist' arbitrary: zs auxlist'' i)
case Nil
then show ?case
by (auto dest!: antisym[OF progress_Until_le])
next
case (Cons a aux')
obtain t a1 a2 where a = (t, a1, a2) by (cases a)
from Cons.prem1(2) have aux': wf_until_auxlist σ V n R pos φ'' I ψ'' aux' (Suc i)
by (rule wf_until_aux_Cons)
from Cons.prem1(2) have 1: t = τ σ i
unfolding ⟨a = (t, a1, a2)⟩ by (rule wf_until_aux_Cons1)
from Cons.prem1(2) have 3: qtable n (Formula.fv ψ'') (mem_restr R) (λv.
(∃ j ≥ i. j < Suc (i + length aux') ∧ mem (τ σ j - τ σ i) I ∧ Formula.sat σ V (map the v) j ψ'' ∧
(∀ k ∈ {i..<j}. if pos then Formula.sat σ V (map the v) k φ'' else ¬ Formula.sat σ V (map the v)
k φ''))) a2
unfolding ⟨a = (t, a1, a2)⟩ by (rule wf_until_aux_Cons3)
from Cons.prem1(3) have Suc_i_aux': Suc i + length aux' =
min (progress σ P' φ''' (Suc j)) (progress σ P' ψ'' (Suc j))
by simp
have i ≥ progress σ P' (Formula.Until φ''' I ψ'') (Suc j)
if enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt) ≤ enat t + right I
using that nts' unfolding wf_ts_def progress.simps
by (auto simp add: 1 list_all2_Cons2 upt_eq_Cons_conv φ'''
intro!: cInf_lower τ_mono elim!: order.trans[rotated] simp del: upt_Suc split: if_splits list.splits)
moreover
have Suc i ≤ progress σ P' (Formula.Until φ''' I ψ'') (Suc j)
if enat t + right I < enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt)
proof -
from that obtain m where m: right I = enat m by (cases right I) auto
have τ_min: τ σ (min j k) = min (τ σ j) (τ σ k) for k
by (simp add: min_of_mono monoI)
have le_progress_iff[simp]: (Suc j) ≤ progress σ P' φ (Suc j) ↔ progress σ P' φ (Suc j) = (Suc
j) for φ
using wf_envs_progress_le[OF MUntil.prem1(2), of φ] by auto
have min_Suc[simp]: min j (Suc j) = j by auto
let ?X = {i. ∀ k. k < Suc j ∧ k ≤ min (progress σ P' φ''' (Suc j)) (progress σ P' ψ'' (Suc j)) →
enat (τ σ k) ≤ enat (τ σ i) + right I}
let ?min = min j (min (progress σ P' φ'' (Suc j)) (progress σ P' ψ'' (Suc j)))
have τ σ ?min ≤ τ σ j
by (rule τ_mono) auto
from m have ?X ≠ {}
by (auto dest!: τ_mono[of _ progress σ P' φ'' (Suc j) σ]

```

```

      simp: not_le not_less φ''' intro!: exI[of _ progress σ P' φ'' (Suc j)]
from m show ?thesis
  using that nts' unfolding wf_ts_def progress.simps
  by (intro cInf_greatest[OF ‹?X ≠ {}›])
    (auto simp: 1 φ''' not_le not_less list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq
      simp del: upt_Suc split: list.splits if_splits
      dest!: spec[of _ ?min] less_le_trans[of τ σ i + m τ σ _ τ σ _ + m] less_τD)
qed
moreover have *: k < progress σ P' ψ (Suc j) if
  enat (τ σ i) + right I < enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt)
  enat (τ σ k - τ σ i) ≤ right I ψ = ψ'' ∨ ψ = φ'' for k ψ
proof -
  from that(1,2) obtain m where right I = enat m
    τ σ i + m < (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt) τ σ k - τ σ i ≤ m
  by (cases right I) auto
  with that(3) nts' progress_le[of σ ψ'' Suc j] progress_le[of σ φ'' Suc j]
  show ?thesis
    unfolding wf_ts_def le_diff_conv
    by (auto simp: not_le list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq add commute
      simp del: upt_Suc split: list.splits if_splits dest!: le_less_trans[of τ σ k] less_τD)
qed
ultimately show ?case using Cons.premis Suc_i_aux'[simplified]
  unfolding ‹a = (t, a1, a2)›
  by (auto simp: φ''' 1 sat.simps upt_conv_Cons dest!: Cons.IH[OF _ aux' Suc_i_aux']
    simp del: upt_Suc split: if_splits prod.splits intro!: iff_exI qtable_cong[OF 3 refl])
qed
thm this
note wf_aux'' = this[OF wf_envs_progress_mono[OF MUntil.premis(2) le_SucI[OF order_refl]]
  wf_auxlist' conjunct2[OF update1, unfolded ne_def length_aux']]
have progress σ P (formula.Until φ''' I ψ'') j + length auxlist' =
  progress σ P' (formula.Until φ''' I ψ'') (Suc j) + length auxlist''
using wf_aux'' valid_aux'' length_aux'_aux''
by (auto simp add: ne_def length_aux' length_aux'')
then have cur =
  (if progress σ P' (formula.Until φ''' I ψ'') (Suc j) + length auxlist'' = 0 then 0
  else τ σ (progress σ P' (formula.Until φ''' I ψ'') (Suc j) + length auxlist'' - 1))
unfolding cur ne_def by auto
then have wf_until_aux σ V R args φ'' ψ'' aux'' (progress σ P' (formula.Until φ''' I ψ'') (Suc j)) ∧
  progress σ P (formula.Until φ''' I ψ'') j + length zs = progress σ P' (formula.Until φ''' I ψ'') (Suc
j) ∧
  progress σ P (formula.Until φ''' I ψ'') j + length zs + length_muaux args aux'' = min (progress σ
P' φ''' (Suc j)) (progress σ P' ψ'' (Suc j)) ∧
  list_all2 (λi. qtable n (fv ψ'') (mem_restr R) (λv. Formula.sat σ V (map the v) i (formula.Until
(if pos then φ'' else formula.Neg φ'') I ψ'')))
  [progress σ P (formula.Until φ''' I ψ'') j..<progress σ P (formula.Until φ''' I ψ'') j + length zs] zs
  using wf_aux'' valid_aux'' fvi_subset
  unfolding wf_until_aux_def length_aux'' args_ivl args_n args_pos by (auto simp only: length_aux'')
}
note update = this
from MUntil.IH(1)[OF φ MUntil.premis(2)] MUntil.IH(2)[OF ψ MUntil.premis(2)] pos pos_eq fvi_subset
show ?case
  by (auto 0 4 simp: args_ivl args_n args_pos Until_eq φ''' progress.simps(6) split: prod.split if_splits
    dest!: update[OF refl refl, rotated]
    intro!: wf_mformula.Until[OF _ _ _ _ args_ivl args_n args_L args_R fvi_subset]
    elim!: list.rel_mono_strong qtable_cong
    elim: mbuf2t_take_add'(1)[OF _ wf_envs_P_simps[OF MUntil.premis(2)] buf nts_snoc]
      mbuf2t_take_add'(2)[OF _ wf_envs_P_simps[OF MUntil.premis(2)] buf nts_snoc])
next

```

```

case (MMatchP I mr mrs  $\varphi$ s buf nts aux)
note sat.simps[simp del] mbufnt_take.simps[simp del] mbufn_add.simps[simp del]
from MMatchP.prem obtain r  $\psi$ s where eq:  $\varphi' = \text{Formula.MatchP } I \ r$ 
  and safe: safe_regex Past Strict r
  and mr: to_mregex r = (mr,  $\psi$ s)
  and mrs: mrs = sorted_list_of_set (RPDs mr)
  and  $\psi$ s: list_all2 (wf_mformula  $\sigma$  j P V n R)  $\varphi$ s  $\psi$ s
  and buf: wf_mbufn'  $\sigma$  P V j n R r buf
  and nts: wf_ts_regex  $\sigma$  P j r nts
  and aux: wf_matchP_aux  $\sigma$  V n R I r aux (progress  $\sigma$  P (Formula.MatchP I r) j)
  by (cases rule: wf_mformula.cases) (auto)
have nts_snoc: list_all2 ( $\lambda i \ t. \ t = \tau \ \sigma \ i$ ) [progress_regex  $\sigma$  P r j..<Suc j] (nts @ [ $\tau \ \sigma \ j$ ])
  using nts unfolding wf_ts_regex_def
  by (auto simp add: wf_envs_progress_regex_le[OF MMatchP.prem(2)] intro: list_all2_appendI)
have update: wf_matchP_aux  $\sigma$  V n R I r (snd (zs, aux')) (progress  $\sigma$  P' (Formula.MatchP I r) (Suc
j))  $\wedge$ 
  list_all2 ( $\lambda i. \ \text{qtable } n \ (\text{Formula.fv\_regex } r) \ (\text{mem\_restr } R)$ 
  ( $\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ (\text{Formula.MatchP } I \ r)$ ))
  [progress  $\sigma$  P (Formula.MatchP I r) j..<progress  $\sigma$  P' (Formula.MatchP I r) (Suc j)] (fst (zs, aux'))
if eval: map (fst o meval n ( $\tau \ \sigma \ j$ ) db)  $\varphi$ s = xss
  and eq: mbufnt_take ( $\lambda \text{rels } t \ (zs, \text{aux})$ .
  case update_matchP n I mr mrs rels t aux of (z, x)  $\Rightarrow$  (zs @ [z], x)
  ([], aux) (mbufn_add xss buf) (nts @ [ $\tau \ \sigma \ j$ ]) = ((zs, aux'), buf', nts')
for xss zs aux' buf' nts'
  unfolding progress_simps
proof (rule mbufnt_take_add_induct'[where j'=Suc j and z'=(zs, aux'), OF eq wf_envs_P_simps[OF
MMatchP.prem(2)] safe mr buf nts_snoc],
  goal_cases xss _ base step)
  case xss
  then show ?case
  using eval  $\psi$ s
  by (auto simp: list_all3_map map2_map_map list_all3_list_all2_conv list_rel_map
  list_rel_flip[symmetric, of _  $\psi$ s  $\varphi$ s] dest!: MMatchP.IH(1)[OF _ _ MMatchP.prem(2)]
  elim!: list_rel_mono_strong split: prod.splits)
next
  case base
  then show ?case
  using aux by auto
next
  case (step k Xs z)
  then show ?case
  by (auto simp: Un_absorb1 mrs safe mr elim!: update_matchP(1) list_all2_appendI
  dest!: update_matchP(2) split: prod.split)
qed simp
then show ?case using  $\psi$ s
  by (auto simp: eq mr mrs safe map_split_alt list_rel_flip[symmetric, of _  $\psi$ s  $\varphi$ s]
  list_all3_map map2_map_map list_all3_list_all2_conv list_rel_map intro!: wf_mformula.intros
  elim!: list_rel_mono_strong mbufnt_take_add'(1)[OF _ wf_envs_P_simps[OF MMatchP.prem(2)]
  safe mr buf nts_snoc]
  mbufnt_take_add'(2)[OF _ wf_envs_P_simps[OF MMatchP.prem(2)] safe mr buf nts_snoc]
  dest!: MMatchP.IH[OF _ _ MMatchP.prem(2)] split: prod.splits)
next
  case (MMatchF I mr mrs  $\varphi$ s buf nts aux)
note sat.simps[simp del] mbufnt_take.simps[simp del] mbufn_add.simps[simp del] progress_simps[simp
del]
from MMatchF.prem obtain r  $\psi$ s where eq:  $\varphi' = \text{Formula.MatchF } I \ r$ 
  and safe: safe_regex Futu Strict r
  and mr: to_mregex r = (mr,  $\psi$ s)

```

```

and mrs: mrs = sorted_list_of_set (LPDs mr)
and  $\psi$ s: list_all2 (wf_mformula  $\sigma$  j P V n R)  $\varphi$ s  $\psi$ s
and buf: wf_mbufn'  $\sigma$  P V j n R r buf
and nts: wf_ts_regex  $\sigma$  P j r nts
and aux: wf_matchF_aux  $\sigma$  V n R I r aux (progress  $\sigma$  P (Formula.MatchF I r) j) 0
and length_aux: progress  $\sigma$  P (Formula.MatchF I r) j + length aux = progress_regex  $\sigma$  P r j
by (cases rule: wf_mformula.cases) (auto)
have nts_snoc: list_all2 ( $\lambda$ i t. t =  $\tau$   $\sigma$  i)
[progress_regex  $\sigma$  P r j..<Suc j] (nts @ [ $\tau$   $\sigma$  j])
using nts unfolding wf_ts_regex_def
by (auto simp add: wf_envs_progress_regex_le[OF MMatchF.prem(2)] intro: list_all2_appendI)
{
fix xss zs aux' aux'' buf' nts'
assume eval: map (fst o meval n ( $\tau$   $\sigma$  j) db)  $\varphi$ s = xss
and eq1: mbufnt_take (update_matchF n I mr mrs) aux (mbufn_add xss buf) (nts @ [ $\tau$   $\sigma$  j]) =
(aux', buf', nts')
and eq2: eval_matchF I mr (case nts' of []  $\Rightarrow$   $\tau$   $\sigma$  j | nt # _  $\Rightarrow$  nt) aux' = (zs, aux'')
have update1: wf_matchF_aux  $\sigma$  V n R I r aux' (progress  $\sigma$  P (Formula.MatchF I r) j) 0  $\wedge$ 
progress  $\sigma$  P (Formula.MatchF I r) j + length aux' = progress_regex  $\sigma$  P' r (Suc j)
using eval nts_snoc nts_snoc length_aux aux  $\psi$ s
by (elim mbufnt_take_add_induct'[where j'=Suc j, OF eq1 wf_envs_P_simps[OF MMatchF.prem(2)]]
safe mr buf)
(auto simp: length_update_matchF
list_all3_map_map2_map_map list_all3_list_all2_conv list.rel_map list.rel_flip[symmetric, of
_  $\psi$ s  $\varphi$ s]
dest!: MMatchF.IH[OF __ MMatchF.prem(2)]
elim: wf_update_matchF[OF safe mr mrs] elim!: list.rel_mono_strong)
from MMatchF.prem(2) have nts': wf_ts_regex  $\sigma$  P' (Suc j) r nts'
using eval eval nts_snoc  $\psi$ s
unfolding wf_ts_regex_def
by (intro mbufnt_take_eqD(2)[OF eq1 wf_mbufn_add[where js'=map ( $\lambda$  $\varphi$ . progress  $\sigma$  P'  $\varphi$  (Suc
j))  $\psi$ s,
OF buf[unfolded wf_mbufn'_def mr prod.case]]])
(auto simp: to_mregex_progress[OF safe mr] Mini_def
list_all3_map_map2_map_map list_all3_list_all2_conv list.rel_map list.rel_flip[symmetric, of
_  $\psi$ s  $\varphi$ s]
list_all2_Cons1 elim!: list.rel_mono_strong intro!: list.rel_refl_strong
dest!: MMatchF.IH[OF __ MMatchF.prem(2)])
have i  $\leq$  progress  $\sigma$  P' (Formula.MatchF I r) (Suc j)  $\implies$ 
wf_matchF_aux  $\sigma$  V n R I r aux' i 0  $\implies$ 
i + length aux' = progress_regex  $\sigma$  P' r (Suc j)  $\implies$ 
wf_matchF_aux  $\sigma$  V n R I r aux'' (progress  $\sigma$  P' (Formula.MatchF I r) (Suc j)) 0  $\wedge$ 
i + length zs = progress  $\sigma$  P' (Formula.MatchF I r) (Suc j)  $\wedge$ 
i + length zs + length aux'' = progress_regex  $\sigma$  P' r (Suc j)  $\wedge$ 
list_all2 ( $\lambda$ i. qtable n (Formula.fv_regex r) (mem_restr R)
( $\lambda$ v. Formula.sat  $\sigma$  V (map the v) i (Formula.MatchF I r)))
[i..<i + length zs] zs for i
using eq2
proof (induction aux' arbitrary: zs aux'' i)
case Nil
then show ?case by (auto dest!: antisym[OF progress_MatchF_le])
next
case (Cons a aux')
obtain t rels rel where a = (t, rels, rel) by (cases a)
from Cons.prem(2) have aux': wf_matchF_aux  $\sigma$  V n R I r aux' (Suc i) 0
by (rule wf_matchF_aux_Cons)
from Cons.prem(2) have 1: t =  $\tau$   $\sigma$  i
unfolding  $\langle$ a = (t, rels, rel) $\rangle$  by (rule wf_matchF_aux_Cons1)

```

from *Cons.prem3*(2) **have** 3: *qtable n (Formula.fv_regex r) (mem_restr R) (λv.*
(∃ j ≥ i. j < Suc (i + length aux') ∧ mem (τ σ j - τ σ i) I ∧ Regex.match (Formula.sat σ V (map
the v)) r i j)) rel
unfolding $\langle a = (t, \text{rels}, \text{rel}) \rangle$ **using** *wf_matchF_aux_Cons3* **by** *force*
from *Cons.prem3*(3) **have** *Suc_i_aux'*: *Suc i + length aux' = progress_regex σ P' r (Suc j)*
by *simp*
have *i ≥ progress σ P' (Formula.MatchF I r) (Suc j)*
if *enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt) ≤ enat t + right I*
using *that nts' unfolding wf_ts_regex_def progress_simps*
by (*auto simp add: 1 list_all2_Cons2 upt_eq_Cons_conv*
intro!: cInf_lower τ_mono elim!: order.trans[rotated] simp del: upt_Suc split: if_splits list.splits)
moreover
have *Suc i ≤ progress σ P' (Formula.MatchF I r) (Suc j)*
if *enat t + right I < enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt)*
proof -
from *that obtain m where m: right I = enat m* **by** (*cases right I*) *auto*
have *τ_min: τ σ (min j k) = min (τ σ j) (τ σ k)* **for** *k*
by (*simp add: min_of_mono monoI*)
have *le_progress_iff[simp]: Suc j ≤ progress σ P' φ (Suc j) ↔ progress σ P' φ (Suc j) = (Suc*
j) for φ
using *wf_envs_progress_le[OF MMatchF.prem3(2), of φ]* **by** *auto*
have *min_Suc[simp]: min j (Suc j) = j* **by** *auto*
let *?X = {i. ∀ k. k < Suc j ∧ k ≤ progress_regex σ P' r (Suc j) → enat (τ σ k) ≤ enat (τ σ i)*
+ right I}
let *?min = min j (progress_regex σ P' r (Suc j))*
have *τ σ ?min ≤ τ σ j*
by (*rule τ_mono*) *auto*
from *m* **have** *?X ≠ {}*
by (*auto dest!: less_τD add_lessD1 simp: not_le not_less*)
from *m* **show** *?thesis*
using *that nts' wf_envs_progress_regex_le[OF MMatchF.prem3(2), of r]*
unfolding *wf_ts_regex_def progress_simps*
by (*intro cInf_greatest[OF ?X ≠ {}]*)
(auto simp: 1 not_le not_less list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq
simp del: upt_Suc split: list.splits if_splits
dest!: spec[of _ ?min] less_le_trans[of τ σ i + m τ σ _ τ σ _ + m] less_τD)
qed
moreover **have** **: k < progress_regex σ P' r (Suc j)* **if**
enat (τ σ i) + right I < enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt)
enat (τ σ k - τ σ i) ≤ right I **for** *k*
proof -
from *that(1,2) obtain m where right I = enat m*
τ σ i + m < (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt) τ σ k - τ σ i ≤ m
by (*cases right I*) *auto*
with *nts' wf_envs_progress_regex_le[OF MMatchF.prem3(2), of r]*
show *?thesis*
unfolding *wf_ts_regex_def le_diff_conv*
by (*auto simp: not_le list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq add commute*
simp del: upt_Suc split: list.splits if_splits dest!: le_less_trans[of τ σ k] less_τD)
qed
ultimately show *?case* **using** *Cons.prem3 Suc_i_aux'[simplified]*
unfolding $\langle a = (t, \text{rels}, \text{rel}) \rangle$
by (*auto simp: 1 sat_simps upt_conv_Cons dest!: Cons.IH[OF _ aux' Suc_i_aux']*
simp del: upt_Suc split: if_splits prod.splits intro!: iff_exI qtable_cong[OF 3 refl])
qed
note *this[OF progress_mono_gen[OF le_SucI, OF order.refl] conjunct1[OF update1] conjunct2[OF*
update1]]

```

}
note update = this[OF refl, rotated]
with MMatchF.premis(2) show ?case using  $\psi$ s
  by (auto simp: eq mr mrs safe map_split_alt list_rel_flip[symmetric, of _  $\psi$ s  $\varphi$ s]
      list_all3_map map2_map_map list_all3_list_all2_conv list_rel_map intro!: wf_mformula.intros
      elim!: list_rel_mono_strong mbufnt_take_add'(1)[OF _ wf_envs_P_simps[OF MMatchF.premis(2)]]
safe mr buf_nts_snoc]
  mbufnt_take_add'(2)[OF _ wf_envs_P_simps[OF MMatchF.premis(2)]] safe mr buf_nts_snoc]
  dest!: MMatchF.IH[OF _ MMatchF.premis(2)] update split: prod.splits)
qed

```

6.6.4 Monitor step

lemma (in *maux*) *wf_mstate_mstep*: $wf_mstate\ \varphi\ \pi\ R\ st \implies last_ts\ \pi \leq snd\ tdb \implies$
 $wf_mstate\ \varphi\ (psnoc\ \pi\ tdb)\ R\ (snd\ (mstep\ (map_prod\ mk_db\ id\ tdb)\ st))$
unfolding *wf_mstate_def* *mstep_def* *Let_def*
by (fastforce simp add: progress_mono le_imp_diff_is_add split: prod.splits
elim!: prefix_of_psnocE dest: meval[OF _ wf_envs_mk_db] list_all2_lengthD)

definition *flatten_verdicts* $Vs = (\bigcup (set (map (\lambda(i, X). (\lambda v. (i, v)) ' X) Vs)))$

lemma *flatten_verdicts_append*[*simp*]:
 $flatten_verdicts\ (Vs\ @\ Us) = flatten_verdicts\ Vs \cup flatten_verdicts\ Us$
by (induct Vs) (auto simp: flatten_verdicts_def)

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 –

from $prefix_of_psnocE[OF\ assms(3,2)]$ **have** $prefix_of\ \pi\ \sigma$
 $\Gamma\ \sigma\ (plen\ \pi) = fst\ tdb\ \tau\ \sigma\ (plen\ \pi) = snd\ tdb$ **by** *auto*
moreover from $assms(1)$ $\langle prefix_of\ \pi\ \sigma \rangle$ **have** $mstate_n\ st = Formula.nfv\ \varphi$
 $mstate_i\ st = progress\ \sigma\ Map.empty\ \varphi\ (plen\ \pi)\ wf_mformula\ \sigma\ (plen\ \pi)\ Map.empty\ Map.empty$
 $(mstate_n\ st)\ R\ (mstate_m\ st)\ \varphi$
unfolding *wf_mstate_def* **by** *blast+*
moreover from $meval[OF\ \langle wf_mformula\ \sigma\ (plen\ \pi)\ Map.empty\ Map.empty\ (mstate_n\ st)\ R\ (mstate_m\ st)\ \varphi \rangle wf_envs_mk_db]$ **obtain** $Vs\ st'$ **where**
 $meval\ (mstate_n\ st)\ (\tau\ \sigma\ (plen\ \pi))\ (mk_db\ (\Gamma\ \sigma\ (plen\ \pi)))\ (mstate_m\ st) = (Vs, st')$
 $wf_mformula\ \sigma\ (Suc\ (plen\ \pi))\ Map.empty\ Map.empty\ (mstate_n\ st)\ R\ st'\ \varphi$
 $list_all2\ (\lambda i. qtable\ (mstate_n\ st)\ (fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ Map.empty\ (map\ the\ v)\ i\ \varphi))$
 $[progress\ \sigma\ Map.empty\ \varphi\ (plen\ \pi)..<progress\ \sigma\ Map.empty\ \varphi\ (Suc\ (plen\ \pi))]\ Vs$ **by** *blast*
moreover from $this\ assms(4)$ **have** $qtable\ (mstate_n\ st)\ (fv\ \varphi)\ (mem_restr\ R)$
 $(\lambda v. Formula.sat\ \sigma\ Map.empty\ (map\ the\ v)\ i\ \varphi)\ (Vs!\ (i - progress\ \sigma\ Map.empty\ \varphi\ (plen\ \pi)))$
if $progress\ \sigma\ Map.empty\ \varphi\ (plen\ \pi) \leq i < progress\ \sigma\ Map.empty\ \varphi\ (Suc\ (plen\ \pi))$
using that **by** (auto simp: list_all2_conv_all_nth
dest!: spec[of _ (i - progress σ Map.empty φ (plen π))])
ultimately show ?thesis
using $assms(4)$ **unfolding** *mstep_def* *Let_def* *flatten_verdicts_def*
by (auto simp: in_set_enumerate_eq list_all2_conv_all_nth progress_mono le_imp_diff_is_add
elim!: in_qtableE in_qtableI intro!: bexI[of _ (i, Vs! (i - progress σ Map.empty φ (plen π)))]))

qed

6.6.5 Monitor function

locale *verimon* = *verimon_spec* + *maux*

lemma (in *verimon*) *mstep_mverdicts*:

assumes *wf*: *wf_mstate* φ π *R* *st*

and *le[simp]*: *last_ts* $\pi \leq \text{snd } \text{tdb}$

and *restrict*: *mem_restr* *R* *v*

shows $(i, v) \in \text{flatten_verdicts } (\text{fst } (\text{mstep } (\text{map_prod } \text{mk_db } \text{id } \text{tdb}) \text{ st})) \longleftrightarrow$
 $(i, v) \in M (\text{psnoc } \pi \text{ tdb}) - M \pi$

proof –

obtain σ **where** *p2*: *prefix_of* (*psnoc* π *tdb*) σ

using *ex_prefix_of* **by** *blast*

with *le* **have** *p1*: *prefix_of* π σ **by** (*blast elim!*: *prefix_of_psnocE*)

show *?thesis*

unfolding *M_def*

by (*auto* 0 3 *simp*: *p2 progress_prefix_conv[OF _ p1] sat_prefix_conv[OF _ p1] not_less*

pprogress_eq[OF p1] pprogress_eq[OF p2]

dest: *mstep_output_iff[OF wf le p2 restrict, THEN iffD1] spec[of _ σ]*

mstep_output_iff[OF wf le _ restrict, THEN iffD1] progress_sat_cong[OF p1]

intro: *mstep_output_iff[OF wf le p2 restrict, THEN iffD2] p1*)

qed

context *maux*

begin

primrec *msteps0* **where**

msteps0 [] *st* = ([], *st*)

| *msteps0* (*tdb* # π) *st* =

(*let* (*V'*, *st'*) = *mstep* (*map_prod* *mk_db* *id* *tdb*) *st*; (*V''*, *st''*) = *msteps0* π *st'* *in* (*V'* @ *V''*, *st''*))

primrec *msteps0_stateless* **where**

msteps0_stateless [] *st* = []

| *msteps0_stateless* (*tdb* # π) *st* = (*let* (*V'*, *st'*) = *mstep* (*map_prod* *mk_db* *id* *tdb*) *st* *in* *V'* @ *msteps0_stateless* π *st'*)

lemma *msteps0_msteps0_stateless*: *fst* (*msteps0* *w* *st*) = *msteps0_stateless* *w* *st*

by (*induct* *w* *arbitrary*: *st*) (*auto simp*: *split_beta*)

lift_definition *msteps* :: *Formula.prefix* \Rightarrow (*'msaux*, *'muaux*) *mstate* \Rightarrow (*nat* \times *event_data* *table*) *list* \times
(*'msaux*, *'muaux*) *mstate*

is *msteps0* .

lift_definition *msteps_stateless* :: *Formula.prefix* \Rightarrow (*'msaux*, *'muaux*) *mstate* \Rightarrow (*nat* \times *event_data* *table*) *list*

is *msteps0_stateless* .

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

by *transfer* (*rule* *msteps0_msteps0_stateless*)

lemma *msteps0_snoc*: *msteps0* (π @ [*tdb*]) *st* =

(*let* (*V'*, *st'*) = *msteps0* π *st*; (*V''*, *st''*) = *mstep* (*map_prod* *mk_db* *id* *tdb*) *st'* *in* (*V'* @ *V''*, *st''*))

by (*induct* π *arbitrary*: *st*) (*auto split*: *prod.splits*)

lemma *msteps_psnoc*: *last_ts* $\pi \leq \text{snd } \text{tdb} \implies$ *msteps* (*psnoc* π *tdb*) *st* =

(*let* (*V'*, *st'*) = *msteps* π *st*; (*V''*, *st''*) = *mstep* (*map_prod* *mk_db* *id* *tdb*) *st'* *in* (*V'* @ *V''*, *st''*))

by *transfer'* (*auto simp*: *msteps0_snoc split*: *list.splits prod.splits if_splits*)

definition *monitor* **where**

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

end

lemma *Suc_length_conv_snoc*: $(\text{Suc } n = \text{length } xs) = (\exists y \text{ ys. } xs = \text{ys} @ [y] \wedge \text{length } \text{ys} = n)$
by (*cases xs rule: rev_cases*) *auto*

lemma (**in** *verimon*) *wf_mstate_msteps*: $wf_mstate \varphi \pi R st \implies mem_restr R v \implies \pi \leq \pi' \implies$
 $X = msteps (pdrop (plen \pi) \pi') st \implies wf_mstate \varphi \pi' R (snd X) \wedge$
 $((i, v) \in flatten_verdicts (fst X)) = ((i, v) \in M \pi' - M \pi)$

proof (*induct plen \pi' - plen \pi arbitrary: X st \pi \pi'*)

case 0

from 0(1,4,5) **have** $\pi = \pi'$ $X = ([], st)$

by (*transfer; auto*)⁺

with 0(2) **show** ?*case unfolding flatten_verdicts_def by simp*

next

case (*Suc x*)

from *Suc(2,5)* **obtain** π'' *tdb* **where** $x = plen \pi'' - plen \pi$ $\pi \leq \pi''$

$\pi' = psnoc \pi''$ *tdb* $pdrop (plen \pi) (psnoc \pi''$ *tdb*) = $psnoc (pdrop (plen \pi) \pi')$ *tdb*

$last_ts (pdrop (plen \pi) \pi') \leq snd$ *tdb* $last_ts \pi'' \leq snd$ *tdb*

$\pi'' \leq psnoc \pi''$ *tdb*

proof (*atomize_elim, transfer, elim exE, goal_cases prefix*)

case (*prefix _ _ \pi' _ \pi_tdb*)

then show ?*case*

proof (*cases \pi_tdb rule: rev_cases*)

case (*snoc \pi_tdb*)

with *prefix show ?thesis*

by (*intro beXI[of _ \pi' @ \pi] exI[of _ tdb]*)

(*force simp: sorted_append append_eq Cons_conv split: list.splits if_splits*)⁺

qed *simp*

qed

with *Suc(1)[OF this(1) Suc.prem(1,2) this(2) refl] Suc.prem show ?case*

unfolding *msteps_msteps_stateless[symmetric]*

by (*auto simp: msteps_psnoc split_beta mstep_mverdicts*

dest: mono_monitor[THEN set_mp, rotated] intro!: wf_mstate_mstep)

qed

lemma (**in** *verimon*) *wf_mstate_msteps_stateless*:

assumes $wf_mstate \varphi \pi R st mem_restr R v \pi \leq \pi'$

shows $(i, v) \in flatten_verdicts (msteps_stateless (pdrop (plen \pi) \pi') st) \iff (i, v) \in M \pi' - M \pi$

using $wf_mstate_msteps[OF assms refl]$ **unfolding** *msteps_msteps_stateless* **by** *simp*

lemma (**in** *verimon*) *wf_mstate_msteps_stateless_UNIV*: $wf_mstate \varphi \pi UNIV st \implies \pi \leq \pi' \implies$

$flatten_verdicts (msteps_stateless (pdrop (plen \pi) \pi') st) = M \pi' - M \pi$

by (*auto dest: wf_mstate_msteps_stateless[OF _ mem_restr_UNIV]*)

lemma (**in** *verimon*) *mverdicts_Nil*: $M pnil = \{\}$

by (*simp add: M_def pprogress_eq*)

context *maux*

begin

lemma *minit_safe_minit*: $mmonitorable \varphi \implies minit_safe \varphi = minit \varphi$

unfolding *minit_safe_def monitorable_formula_code* **by** *simp*

lemma *wf_mstate_minit_safe*: $mmonitorable \varphi \implies wf_mstate \varphi pnil R (minit_safe \varphi)$

using $wf_mstate_minit minit_safe_minit mmonitorable_def$ **by** *metis*

end

lemma (**in** *verimon*) *monitor_mverdicts*: $flatten_verdicts (monitor \varphi \pi) = M \pi$

```

unfolding monitor_def using monitorable
by (subst wf_mstate_msteps_stateless_UNIV[OF wf_mstate_init_safe, simplified])
(auto simp: mmonitorable_def mverdicts_Nil)

```

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 *init_safe* and *mstep* preserves the invariant *wf_mstate* and produces the the verdicts according to M :
 - *wf_mstate_init_safe*: $\text{mmonitorable } \varphi' \implies \text{wf_mstate } \varphi' \text{pnil } R (\text{init_safe } \varphi')$
 - *wf_mstate_mstep*: $\llbracket \text{wf_mstate } \varphi' \pi R \text{st}; \text{last_ts } \pi \leq \text{snd } \text{tdb} \rrbracket \implies \text{wf_mstate } \varphi' (\text{psnoc } \pi \text{tdb}) R (\text{snd } (\text{mstep } (\text{map_prod } \text{mk_db } \text{id } \text{tdb}) \text{st}))$
 - *mstep_mverdicts*: $\llbracket \text{wf_mstate } \varphi \pi R \text{st}; \text{last_ts } \pi \leq \text{snd } \text{tdb}; \text{mem_restr } R v \rrbracket \implies ((i, v) \in \text{flatten_verdicts } (\text{fst } (\text{mstep } (\text{map_prod } \text{mk_db } \text{id } \text{tdb}) \text{st}))) = ((i, v) \in M (\text{psnoc } \pi \text{tdb}) - M \pi)$
3. The executable monitor's offline interface *local.monitor* implements M :
 - *monitor_mverdicts*: $\text{flatten_verdicts } (\text{local.monitor } \varphi \pi) = M \pi$

```

end

```

7 Efficient implementation of temporal operators

7.1 Optimized queue data structure

```

lemma less_enat_iff:  $a < \text{enat } i \iff (\exists j. a = \text{enat } j \wedge j < i)$ 
by (cases a) auto

```

```

type_synonym 'a queue_t = 'a list  $\times$  'a list

```

```

definition queue_invariant :: 'a queue_t  $\Rightarrow$  bool where
queue_invariant  $q = (\text{case } q \text{ of } ([], []) \Rightarrow \text{True} \mid (fs, l \# ls) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False})$ 

```

```

typedef 'a queue = { $q :: 'a \text{ queue\_t}. \text{queue\_invariant } q$ }
by (auto simp: queue_invariant_def split: list.splits)

```

```

setup_lifting type_definition_queue

```

lift_definition *linearize* :: 'a queue \Rightarrow 'a list is $(\lambda q. \text{case } q \text{ of } (fs, ls) \Rightarrow fs @ \text{rev } ls)$.

lift_definition *empty_queue* :: 'a queue is $([], [])$
by (auto simp: queue_invariant_def split: list.splits)

lemma *empty_queue_rep*: *linearize empty_queue* = []
by transfer (simp add: empty_queue_def linearize_def)

lift_definition *is_empty* :: 'a queue \Rightarrow bool is $\lambda q. (\text{case } q \text{ of } ([], []) \Rightarrow \text{True} \mid _ \Rightarrow \text{False})$.

lemma *linearize_t_Nil*: $(\text{case } q \text{ of } (fs, ls) \Rightarrow fs @ \text{rev } ls) = [] \iff q = ([], [])$
by (auto split: prod.splits)

lemma *is_empty_alt*: *is_empty* $q \iff \text{linearize } q = []$
by transfer (auto simp: linearize_t_Nil list.case_eq_if)

fun *prepend_queue_t* :: 'a \Rightarrow 'a queue_t \Rightarrow 'a queue_t **where**
prepend_queue_t a $([], []) = ([], [a])$
| *prepend_queue_t* a $(fs, l \# ls) = (a \# fs, l \# ls)$
| *prepend_queue_t* a $(f \# fs, []) = \text{undefined}$

lift_definition *prepend_queue* :: 'a \Rightarrow 'a queue \Rightarrow 'a queue is *prepend_queue_t*
by (auto simp: queue_invariant_def split: list.splits elim: prepend_queue_t.elims)

lemma *prepend_queue_rep*: *linearize (prepend_queue a q)* = $a \# \text{linearize } q$
by transfer
(auto simp add: queue_invariant_def linearize_def elim: prepend_queue_t.elims split: prod.splits)

lift_definition *append_queue* :: 'a \Rightarrow 'a queue \Rightarrow 'a queue is
 $(\lambda a q. \text{case } q \text{ of } (fs, ls) \Rightarrow (fs, a \# ls))$
by (auto simp: queue_invariant_def split: list.splits)

lemma *append_queue_rep*: *linearize (append_queue a q)* = *linearize* $q @ [a]$
by transfer (auto simp add: linearize_def split: prod.splits)

fun *safe_last_t* :: 'a queue_t \Rightarrow 'a option \times 'a queue_t **where**
safe_last_t $([], []) = (\text{None}, ([], []))$
| *safe_last_t* $(fs, l \# ls) = (\text{Some } l, (fs, l \# ls))$
| *safe_last_t* $(f \# fs, []) = \text{undefined}$

lift_definition *safe_last* :: 'a queue \Rightarrow 'a option \times 'a queue is *safe_last_t*
by (auto simp: queue_invariant_def split: prod.splits list.splits)

lemma *safe_last_rep*: *safe_last* $q = (\alpha, q') \implies \text{linearize } q = \text{linearize } q' \wedge$
 $(\text{case } \alpha \text{ of } \text{None} \Rightarrow \text{linearize } q = [] \mid \text{Some } a \Rightarrow \text{linearize } q \neq [] \wedge a = \text{last } (\text{linearize } q))$
by transfer (auto simp: queue_invariant_def split: list.splits elim: safe_last_t.elims)

fun *safe_hd_t* :: 'a queue_t \Rightarrow 'a option \times 'a queue_t **where**
safe_hd_t $([], []) = (\text{None}, ([], []))$
| *safe_hd_t* $([], [l]) = (\text{Some } l, ([], [l]))$
| *safe_hd_t* $([], l \# ls) = (\text{let } fs = \text{rev } ls \text{ in } (\text{Some } (\text{hd } fs), (fs, [l])))$
| *safe_hd_t* $(f \# fs, l \# ls) = (\text{Some } f, (f \# fs, l \# ls))$
| *safe_hd_t* $(f \# fs, []) = \text{undefined}$

lift_definition(code_dt) *safe_hd* :: 'a queue \Rightarrow 'a option \times 'a queue is *safe_hd_t*
proof –

fix $q :: 'a \text{ queue}_t$
assume *queue_invariant* q

then show $\text{pred_prod} \top \text{queue_invariant} (\text{safe_hd_t } q)$
by (*cases* q *rule*: safe_hd_t.cases) (*auto simp*: $\text{queue_invariant_def Let_def split: list.split}$)
qed

lemma safe_hd_rep : $\text{safe_hd } q = (\alpha, q') \implies \text{linearize } q = \text{linearize } q' \wedge$
(case α *of* $\text{None} \implies \text{linearize } q = [] \mid \text{Some } a \implies \text{linearize } q \neq [] \wedge a = \text{hd} (\text{linearize } q)$ *)*
by *transfer*
(auto simp add: $\text{queue_invariant_def Let_def hd_append split: list.splits elim: safe_hd_t.elims}$ *)*

fun replace_hd_t :: $'a \Rightarrow 'a \text{ queue_t} \Rightarrow 'a \text{ queue_t}$ **where**
 $\text{replace_hd_t } a ([], []) = ([], [])$
 $\mid \text{replace_hd_t } a ([], [l]) = ([], [a])$
 $\mid \text{replace_hd_t } a ([], l \# ls) = (\text{let } fs = \text{rev } ls \text{ in } (a \# \text{tl } fs, [l]))$
 $\mid \text{replace_hd_t } a (f \# fs, l \# ls) = (a \# fs, l \# ls)$
 $\mid \text{replace_hd_t } a (f \# fs, []) = \text{undefined}$

lift_definition replace_hd :: $'a \Rightarrow 'a \text{ queue} \Rightarrow 'a \text{ queue}$ **is** replace_hd_t
by (*auto simp*: $\text{queue_invariant_def split: list.splits elim: replace_hd_t.elims}$)

lemma tl_append : $xs \neq [] \implies \text{tl } xs @ ys = \text{tl} (xs @ ys)$
by *simp*

lemma replace_hd_rep : $\text{linearize } q = f \# fs \implies \text{linearize} (\text{replace_hd } a q) = a \# fs$
proof (*transfer fixing*: $f fs a$)

fix q
assume $\text{queue_invariant } q$ **and** *(case* q *of* $(fs, ls) \Rightarrow fs @ \text{rev } ls = f \# fs$ *)*
then show *(case* $\text{replace_hd_t } a q$ *of* $(fs, ls) \Rightarrow fs @ \text{rev } ls = a \# fs$ *)*
by (*cases* (a, q) *rule*: $\text{replace_hd_t.cases}$) (*auto simp*: $\text{queue_invariant_def tl_append}$)
qed

fun replace_last_t :: $'a \Rightarrow 'a \text{ queue_t} \Rightarrow 'a \text{ queue_t}$ **where**
 $\text{replace_last_t } a ([], []) = ([], [])$
 $\mid \text{replace_last_t } a (fs, l \# ls) = (fs, a \# ls)$
 $\mid \text{replace_last_t } a (fs, []) = \text{undefined}$

lift_definition replace_last :: $'a \Rightarrow 'a \text{ queue} \Rightarrow 'a \text{ queue}$ **is** replace_last_t
by (*auto simp*: $\text{queue_invariant_def split: list.splits elim: replace_last_t.elims}$)

lemma replace_last_rep : $\text{linearize } q = fs @ [f] \implies \text{linearize} (\text{replace_last } a q) = fs @ [a]$
by *transfer* (*auto simp*: $\text{queue_invariant_def split: list.splits prod.splits elim!: replace_last_t.elims}$)

fun tl_queue_t :: $'a \text{ queue_t} \Rightarrow 'a \text{ queue_t}$ **where**
 $\text{tl_queue_t} ([], []) = ([], [])$
 $\mid \text{tl_queue_t} ([], [l]) = ([], [])$
 $\mid \text{tl_queue_t} ([], l \# ls) = (\text{tl} (\text{rev } ls), [l])$
 $\mid \text{tl_queue_t} (a \# as, fs) = (as, fs)$

lift_definition tl_queue :: $'a \text{ queue} \Rightarrow 'a \text{ queue}$ **is** tl_queue_t
by (*auto simp*: $\text{queue_invariant_def split: list.splits elim!: tl_queue_t.elims}$)

lemma tl_queue_rep : $\neg \text{is_empty } q \implies \text{linearize} (\text{tl_queue } q) = \text{tl} (\text{linearize } q)$
by *transfer* (*auto simp*: $\text{tl_append split: prod.splits list.splits elim!: tl_queue_t.elims}$)

lemma $\text{length_tl_queue_rep}$: $\neg \text{is_empty } q \implies$
 $\text{length} (\text{linearize} (\text{tl_queue } q)) < \text{length} (\text{linearize } q)$
by *transfer* (*auto split*: $\text{prod.splits list.splits elim: tl_queue_t.elims}$)

lemma $\text{length_tl_queue_safe_hd}$:

```

assumes safe_hd q = (Some a, q')
shows length (linearize (tl_queue q')) < length (linearize q)
using safe_hd_rep[OF assms]
by (auto simp add: length_tl_queue_rep is_empty_alt)

function dropWhile_queue :: ('a ⇒ bool) ⇒ 'a queue ⇒ 'a queue where
  dropWhile_queue f q = (case safe_hd q of (None, q') ⇒ q'
    | (Some a, q') ⇒ if f a then dropWhile_queue f (tl_queue q') else q')
by pat_completeness auto
termination
using length_tl_queue_safe_hd[OF sym]
by (relation measure (λ(f, q). length (linearize q))) (fastforce split: prod.splits)+

lemma dropWhile_hd_tl: xs ≠ [] ⇒
  dropWhile P xs = (if P (hd xs) then dropWhile P (tl xs) else xs)
by (cases xs) auto

lemma dropWhile_queue_rep: linearize (dropWhile_queue f q) = dropWhile f (linearize q)
by (induction f q rule: dropWhile_queue.induct)
  (auto simp add: tl_queue_rep dropWhile_hd_tl is_empty_alt
    split: prod.splits option.splits dest: safe_hd_rep)

function takeWhile_queue :: ('a ⇒ bool) ⇒ 'a queue ⇒ 'a queue where
  takeWhile_queue f q = (case safe_hd q of (None, q') ⇒ q'
    | (Some a, q') ⇒ if f a
      then prepend_queue a (takeWhile_queue f (tl_queue q'))
      else empty_queue)
by pat_completeness auto
termination
using length_tl_queue_safe_hd[OF sym]
by (relation measure (λ(f, q). length (linearize q))) (fastforce split: prod.splits)+

lemma takeWhile_hd_tl: xs ≠ [] ⇒
  takeWhile P xs = (if P (hd xs) then hd xs # takeWhile P (tl xs) else [])
by (cases xs) auto

lemma takeWhile_queue_rep: linearize (takeWhile_queue f q) = takeWhile f (linearize q)
by (induction f q rule: takeWhile_queue.induct)
  (auto simp add: prepend_queue_rep tl_queue_rep empty_queue_rep takeWhile_hd_tl is_empty_alt
    split: prod.splits option.splits dest: safe_hd_rep)

function takedropWhile_queue :: ('a ⇒ bool) ⇒ 'a queue ⇒ 'a queue × 'a list where
  takedropWhile_queue f q = (case safe_hd q of (None, q') ⇒ (q', [])
    | (Some a, q') ⇒ if f a
      then (case takedropWhile_queue f (tl_queue q') of (q'', as) ⇒ (q'', a # as))
      else (q', []))
by pat_completeness auto
termination
using length_tl_queue_safe_hd[OF sym]
by (relation measure (λ(f, q). length (linearize q))) (fastforce split: prod.splits)+

lemma takedropWhile_queue_fst: fst (takedropWhile_queue f q) = dropWhile_queue f q
proof (induction f q rule: takedropWhile_queue.induct)
case (1 f q)
then show ?case
  by (simp split: prod.splits) (auto simp add: case_prod_unfold split: option.splits)
qed

```

lemma *takedropWhile_queue_snd*: $\text{snd} (\text{takedropWhile_queue } f \ q) = \text{takeWhile } f \ (\text{linearize } q)$
proof (*induction* $f \ q$ *rule*: *takedropWhile_queue.induct*)
case ($1 \ f \ q$)
then show *?case*
by (*simp split*: *prod.splits*)
(auto simp add: case_prod_unfold tl_queue_rep takeWhile_hd_tl is_empty_alt
split: option.splits dest: safe_hd_rep)
qed

7.2 Optimized data structure for Since

type_synonym *'a mmsaux* = $ts \times ts \times \text{bool list} \times \text{bool list} \times$
 $(ts \times 'a \text{ table}) \text{ queue} \times (ts \times 'a \text{ table}) \text{ queue} \times$
 $(('a \text{ tuple}, ts) \text{ mapping}) \times (('a \text{ tuple}, ts) \text{ mapping})$

fun *time_mmsaux* :: *'a mmsaux* \Rightarrow *ts* **where**
time_mmsaux aux = $(\text{case } aux \text{ of } (nt, _) \Rightarrow nt)$

definition *ts_tuple_rel* :: $(ts \times 'a \text{ table}) \text{ set} \Rightarrow (ts \times 'a \text{ tuple}) \text{ set}$ **where**
ts_tuple_rel ys = $\{(t, as). \exists X. as \in X \wedge (t, X) \in ys\}$

lemma *finite fst_ts_tuple_rel*: $\text{finite } (\text{fst } ' \{tas \in ts_tuple_rel \ (set \ xs). \ P \ tas\})$

proof –
have $\text{fst } ' \{tas \in ts_tuple_rel \ (set \ xs). \ P \ tas\} \subseteq \text{fst } ' \ ts_tuple_rel \ (set \ xs)$
by *auto*
moreover have $\dots \subseteq \text{set } (\text{map } \text{fst } \ xs)$
by (*force simp add: ts_tuple_rel_def*)
finally show *?thesis*
using *finite_subset* **by** *blast*
qed

lemma *ts_tuple_rel_ext_Cons*: $tas \in ts_tuple_rel \ \{(nt, X)\} \Longrightarrow$
 $tas \in ts_tuple_rel \ (set \ ((nt, X) \# \ tass))$
by (*auto simp add: ts_tuple_rel_def*)

lemma *ts_tuple_rel_ext_Cons'*: $tas \in ts_tuple_rel \ (set \ tass) \Longrightarrow$
 $tas \in ts_tuple_rel \ (set \ ((nt, X) \# \ tass))$
by (*auto simp add: ts_tuple_rel_def*)

lemma *ts_tuple_rel_intro*: $as \in X \Longrightarrow (t, X) \in ys \Longrightarrow (t, as) \in ts_tuple_rel \ ys$
by (*auto simp add: ts_tuple_rel_def*)

lemma *ts_tuple_rel_dest*: $(t, as) \in ts_tuple_rel \ ys \Longrightarrow \exists X. (t, X) \in ys \wedge as \in X$
by (*auto simp add: ts_tuple_rel_def*)

lemma *ts_tuple_rel_Un*: $ts_tuple_rel \ (ys \cup zs) = ts_tuple_rel \ ys \cup ts_tuple_rel \ zs$
by (*auto simp add: ts_tuple_rel_def*)

lemma *ts_tuple_rel_ext*: $tas \in ts_tuple_rel \ \{(nt, X)\} \Longrightarrow$
 $tas \in ts_tuple_rel \ (set \ ((nt, Y \cup X) \# \ tass))$

proof –
assume *assm*: $tas \in ts_tuple_rel \ \{(nt, X)\}$
then obtain *as* **where** *tas_def*: $tas = (nt, as)$ $as \in X$
by (*cases tas*) (*auto simp add: ts_tuple_rel_def*)
then have $as \in Y \cup X$
by *auto*
then show $tas \in ts_tuple_rel \ (set \ ((nt, Y \cup X) \# \ tass))$
unfolding *tas_def*(1)

by (rule ts_tuple_rel_intro) auto
qed

lemma ts_tuple_rel_ext': $tas \in ts_tuple_rel (set ((nt, X) \# tass)) \implies$
 $tas \in ts_tuple_rel (set ((nt, X \cup Y) \# tass))$

proof –
assume *assm*: $tas \in ts_tuple_rel (set ((nt, X) \# tass))$
then have $tas \in ts_tuple_rel \{(nt, X)\} \cup ts_tuple_rel (set tass)$
using ts_tuple_rel_Un by force
then show $tas \in ts_tuple_rel (set ((nt, X \cup Y) \# tass))$
proof
assume $tas \in ts_tuple_rel \{(nt, X)\}$
then show ?thesis
by (auto simp: Un_commute dest!: ts_tuple_rel_ext)
next
assume $tas \in ts_tuple_rel (set tass)$
then have $tas \in ts_tuple_rel (set ((nt, X \cup Y) \# tass))$
by (rule ts_tuple_rel_ext_Cons')
then show ?thesis by simp
qed

qed

lemma ts_tuple_rel_mono: $ys \subseteq zs \implies ts_tuple_rel\ ys \subseteq ts_tuple_rel\ zs$
by (auto simp add: ts_tuple_rel_def)

lemma ts_tuple_rel_filter: $ts_tuple_rel (set (filter (\lambda(t, X). P t) xs)) =$
 $\{(t, X) \in ts_tuple_rel (set xs). P t\}$
by (auto simp add: ts_tuple_rel_def)

lemma ts_tuple_rel_set_filter: $x \in ts_tuple_rel (set (filter P xs)) \implies$
 $x \in ts_tuple_rel (set xs)$
by (auto simp add: ts_tuple_rel_def)

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 \longleftrightarrow X = {}
by (simp add: safe_max_def)

lemma safe_max_empty_dest: safe_max X = None \implies X = {}
by (simp add: safe_max_def split: if_splits)

lemma safe_max_Some_intro: $x \in X \implies \exists y. safe_max\ X = Some\ y$
using safe_max_empty by auto

lemma safe_max_Some_dest_in: finite X $\implies safe_max\ X = Some\ x \implies x \in X$
using Max_in by (auto simp add: safe_max_def split: if_splits)

lemma safe_max_Some_dest_le: finite X $\implies safe_max\ X = Some\ x \implies y \in X \implies y \leq x$
using Max_ge by (auto simp add: safe_max_def split: if_splits)

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 \longleftrightarrow
(args_L args) \subseteq (args_R args) \wedge

$maskL = join_mask (args_n args) (args_L args) \wedge$
 $maskR = join_mask (args_n args) (args_R args) \wedge$
 $(\forall (t, X) \in set\ ys. table (args_n args) (args_R args) X) \wedge$
 $table (args_n args) (args_R args) (Mapping.keys\ tuple_in) \wedge$
 $table (args_n args) (args_R args) (Mapping.keys\ tuple_since) \wedge$
 $(\forall as \in \bigcup (snd\ ' (set (linearize\ data_prev))). wf_tuple (args_n args) (args_R args) as) \wedge$
 $cur = nt \wedge$
 $ts_tuple_rel (set\ ys) =$
 $\{tas \in ts_tuple_rel (set (linearize\ data_prev) \cup set (linearize\ data_in)).$
 $valid_tuple\ tuple_since\ tas\} \wedge$
 $sorted (map\ fst (linearize\ data_prev)) \wedge$
 $(\forall t \in fst\ ' set (linearize\ data_prev). t \leq nt \wedge nt - t < left (args_ivl\ args)) \wedge$
 $sorted (map\ fst (linearize\ data_in)) \wedge$
 $(\forall t \in fst\ ' set (linearize\ data_in). t \leq nt \wedge nt - t \geq left (args_ivl\ args)) \wedge$
 $(\forall as. Mapping.lookup\ tuple_in\ as = safe_max (fst\ '$
 $\{tas \in ts_tuple_rel (set (linearize\ data_in)). valid_tuple\ tuple_since\ tas \wedge as = snd\ tas\})) \wedge$
 $(\forall as \in Mapping.keys\ tuple_since. case\ Mapping.lookup\ tuple_since\ as\ of\ Some\ t \Rightarrow t \leq nt)$

lemma *Mapping_lookup_filter_keys*: $k \in Mapping.keys (Mapping.filter\ f\ m) \Longrightarrow$
 $Mapping.lookup (Mapping.filter\ f\ m)\ k = Mapping.lookup\ m\ k$
by (*metis default_def insert_subset keys_default keys_filter lookup_default lookup_default_filter*)

lemma *Mapping_filter_keys*: $(\forall k \in Mapping.keys\ m. P (Mapping.lookup\ m\ k)) \Longrightarrow$
 $(\forall k \in Mapping.keys (Mapping.filter\ f\ m). P (Mapping.lookup (Mapping.filter\ f\ m)\ k))$
using *Mapping_lookup_filter_keys Mapping.keys_filter* **by** *fastforce*

lemma *Mapping_filter_keys_le*: $(\bigwedge x. P\ x \Longrightarrow P'\ x) \Longrightarrow$
 $(\forall k \in Mapping.keys\ m. P (Mapping.lookup\ m\ k)) \Longrightarrow (\forall k \in Mapping.keys\ m. P' (Mapping.lookup\ m\ k))$
by *auto*

lemma *Mapping_keys_dest*: $x \in Mapping.keys\ f \Longrightarrow \exists y. Mapping.lookup\ f\ x = Some\ y$
by (*simp add: domD keys_dom_lookup*)

lemma *Mapping_keys_intro*: $Mapping.lookup\ f\ x \neq None \Longrightarrow x \in Mapping.keys\ f$
by (*simp add: domIff keys_dom_lookup*)

lemma *valid_mmsaux_tuple_in_keys*: $valid_mmsaux\ args\ cur$
 $(nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since)\ ys \Longrightarrow$
 $Mapping.keys\ tuple_in = snd\ ' \{tas \in ts_tuple_rel (set (linearize\ data_in)).$
 $valid_tuple\ tuple_since\ tas\}$
by (*auto intro!*: *Mapping_keys_intro safe_max_Some_intro*
 $dest!$: *Mapping_keys_dest safe_max_Some_dest_in[OF finite_fst_ts_tuple_rel]*) $+$

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*: $L \subseteq R \Longrightarrow valid_mmsaux (init_args\ I\ n\ L\ R\ b)\ 0$
 $(init_mmsaux (init_args\ I\ n\ L\ R\ b))\ []$
by (*auto simp add: init_args_def empty_queue_rep ts_tuple_rel_def join_mask_def*
 $Mapping.lookup_empty safe_max_def table_def$)

abbreviation *filter_cond* $X'\ ts\ t' \equiv (\lambda as\ _. \neg (as \in X' \wedge Mapping.lookup\ ts\ as = Some\ t'))$

lemma *dropWhile_filter*:
 $sorted (map\ fst\ xs) \Longrightarrow \forall t \in fst\ ' set\ xs. t \leq nt \Longrightarrow$
 $dropWhile (\lambda(t, X). enat (nt - t) > c)\ xs = filter (\lambda(t, X). enat (nt - t) \leq c)\ xs$

by (induction xs) (auto 0 3 intro!: filter_id_conv[THEN iffD2, symmetric] elim: order.trans[rotated])

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

by (induction xs) (auto 0 3 intro!: filter_id_conv[THEN iffD2, symmetric] elim: order.trans[rotated])

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

by (induction xs) (auto 0 3 intro!: filter_id_conv[THEN iffD2, symmetric] elim: order.trans[rotated])

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

by (induction xs) (auto 0 3 simp: less_enat_iff intro!: filter_empty_conv[THEN iffD2, symmetric])

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

by (induction xs) (auto 0 3 simp: less_enat_iff intro!: filter_empty_conv[THEN iffD2, symmetric])

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

by (induction xs) (auto 0 3 simp: less_enat_iff intro!: filter_empty_conv[THEN iffD2, symmetric])

lemma fold_Mapping_filter_None: Mapping.lookup ts as = None \implies

Mapping.lookup (fold ($\lambda(t, X) ts. \text{Mapping.filter}$

(filter_cond X ts t) ts) ds ts) as = None

by (induction ds arbitrary: ts) (auto simp add: Mapping.lookup_filter)

lemma Mapping_lookup_filter_Some_P: Mapping.lookup (Mapping.filter P m) k = Some v $\implies P k v$

by (auto simp add: Mapping.lookup_filter split: option.splits if_splits)

lemma Mapping_lookup_filter_None: ($\bigwedge v. \neg P k v$) \implies

Mapping.lookup (Mapping.filter P m) k = None

by (auto simp add: Mapping.lookup_filter split: option.splits)

lemma Mapping_lookup_filter_Some: ($\bigwedge v. P k v$) \implies

Mapping.lookup (Mapping.filter P m) k = Mapping.lookup m k

by (auto simp add: Mapping.lookup_filter split: option.splits)

lemma Mapping_lookup_filter_not_None: Mapping.lookup (Mapping.filter P m) k \neq None \implies

Mapping.lookup (Mapping.filter P m) k = Mapping.lookup m k

by (auto simp add: Mapping.lookup_filter split: option.splits)

lemma fold_Mapping_filter_Some_None: Mapping.lookup ts as = Some t \implies

as $\in X \implies (t, X) \in \text{set ds} \implies$

Mapping.lookup (fold ($\lambda(t, X) ts. \text{Mapping.filter}$ (filter_cond X ts t) ts) ds ts) as = None

proof (induction ds arbitrary: ts)

case (Cons a ds)

show ?case

proof (cases a)

case (Pair t' X')

with Cons show ?thesis

```

using fold_Mapping_filter_None[of Mapping.filter (filter_cond X' ts t') ts as ds]
      Mapping_lookup_filter_not_None[of filter_cond X' ts t' ts as]
      fold_Mapping_filter_None[OF Mapping_lookup_filter_None, of _ as ds ts]
by (cases Mapping.lookup (Mapping.filter (filter_cond X' ts t') ts) as = None) auto
qed
qed simp

lemma fold_Mapping_filter_Some_Some: Mapping.lookup ts as = Some t  $\implies$ 
  ( $\bigwedge X. (t, X) \in \text{set } ds \implies as \notin X$ )  $\implies$ 
  Mapping.lookup (fold ( $\lambda(t, X)$  ts. Mapping.filter (filter_cond X ts t) ts) ds ts) as = Some t
proof (induction ds arbitrary: ts)
case (Cons a ds)
then show ?case
proof (cases a)
case (Pair t' X')
with Cons show ?thesis
using Mapping_lookup_filter_Some[of filter_cond X' ts t' as ts] by auto
qed
qed simp

fun shift_end :: args  $\Rightarrow$  ts  $\Rightarrow$  'a mmsaux  $\Rightarrow$  'a mmsaux where
  shift_end args nt (t, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since) =
    (let I = args_ivl args;
      data_prev' = dropWhile_queue ( $\lambda(t, X)$ . enat (nt - t) > right I) data_prev;
      (data_in, discard) = takedropWhile_queue ( $\lambda(t, X)$ . enat (nt - t) > right I) data_in;
      tuple_in = fold ( $\lambda(t, X)$  tuple_in. Mapping.filter
        (filter_cond X tuple_in t) tuple_in) discard tuple_in in
    (t, gc, maskL, maskR, data_prev', data_in, tuple_in, tuple_since))

lemma valid_shift_end_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 cur (shift_end args nt
  (ot, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))
  (filter ( $\lambda(t, rel)$ . enat (nt - t)  $\leq$  right (args_ivl args)) auxlist)
proof -
define I where I = args_ivl args
define data_in' where data_in'  $\equiv$ 
  fst (takedropWhile_queue ( $\lambda(t, X)$ . enat (nt - t) > right I) data_in)
define data_prev' where data_prev'  $\equiv$ 
  dropWhile_queue ( $\lambda(t, X)$ . enat (nt - t) > right I) data_prev
define discard where discard  $\equiv$ 
  snd (takedropWhile_queue ( $\lambda(t, X)$ . enat (nt - t) > right I) data_in)
define tuple_in' where tuple_in'  $\equiv$  fold ( $\lambda(t, X)$  tuple_in. Mapping.filter
  ( $\lambda as \_. \neg(as \in X \wedge \text{Mapping.lookup tuple\_in } as = \text{Some } t)$ ) tuple_in) discard tuple_in
have tuple_in_Some_None:  $\bigwedge as t X. \text{Mapping.lookup tuple\_in } as = \text{Some } t \implies$ 
   $as \in X \implies (t, X) \in \text{set } discard \implies \text{Mapping.lookup tuple\_in'} as = \text{None}$ 
using fold_Mapping_filter_Some_None unfolding tuple_in'_def by fastforce
have tuple_in_Some_Some:  $\bigwedge as t. \text{Mapping.lookup tuple\_in } as = \text{Some } t \implies$ 
  ( $\bigwedge X. (t, X) \in \text{set } discard \implies as \notin X$ )  $\implies \text{Mapping.lookup tuple\_in'} as = \text{Some } t$ 
using fold_Mapping_filter_Some_Some unfolding tuple_in'_def by fastforce
have tuple_in_None_None:  $\bigwedge as. \text{Mapping.lookup tuple\_in } as = \text{None} \implies$ 
  Mapping.lookup tuple_in' as = None
using fold_Mapping_filter_None unfolding tuple_in'_def by fastforce
have tuple_in'_keys:  $\bigwedge as. as \in \text{Mapping.keys tuple\_in'} \implies as \in \text{Mapping.keys tuple\_in}$ 
using tuple_in_Some_None tuple_in_Some_Some tuple_in_None_None
by (fastforce intro: Mapping_keys_intro dest: Mapping_keys_dest)

```

```

have F1: sorted (map fst (linearize data_in))  $\forall t \in \text{fst ' set (linearize data_in). } t \leq nt$ 
  using valid_before nt_mono by auto
have F2: sorted (map fst (linearize data_prev))  $\forall t \in \text{fst ' set (linearize data_prev). } t \leq nt$ 
  using valid_before nt_mono by auto
have lin_data_in': linearize data_in' =
  filter ( $\lambda(t, X). \text{enat } (nt - t) \leq \text{right } I$ ) (linearize data_in)
  unfolding data_in'_def[unfolded takedropWhile_queue_fst] dropWhile_queue_rep
  dropWhile_filter[OF F1] ..
then have set_lin_data_in': set (linearize data_in')  $\subseteq$  set (linearize data_in)
  by auto
have sorted (map fst (linearize data_in))
  using valid_before by auto
then have sorted_lin_data_in': sorted (map fst (linearize data_in'))
  unfolding lin_data_in' using sorted_filter by auto
have discard_alt: discard = filter ( $\lambda(t, X). \text{enat } (nt - t) > \text{right } I$ ) (linearize data_in)
  unfolding discard_def[unfolded takedropWhile_queue_snd] takeWhile_filter[OF F1] ..
have lin_data_prev': linearize data_prev' =
  filter ( $\lambda(t, X). \text{enat } (nt - t) \leq \text{right } I$ ) (linearize data_prev)
  unfolding data_prev'_def[unfolded takedropWhile_queue_fst] dropWhile_queue_rep
  dropWhile_filter[OF F2] ..
have sorted (map fst (linearize data_prev))
  using valid_before by auto
then have sorted_lin_data_prev': sorted (map fst (linearize data_prev'))
  unfolding lin_data_prev' using sorted_filter by auto
have lookup_tuple_in':  $\bigwedge as. \text{Mapping.lookup tuple\_in' as} = \text{safe\_max (fst ' } \{tas \in ts\_tuple\_rel (\text{set (linearize data\_in')}). \text{valid\_tuple tuple\_since } tas \wedge as = \text{snd } tas\}$ )
proof -
  fix as
  show 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})
  proof (cases Mapping.lookup tuple_in as)
    case None
      then have {tas  $\in$  ts_tuple_rel (set (linearize data_in)).
        valid_tuple tuple_since tas  $\wedge$  as = snd tas} = {}
        using valid_before by (auto dest!: safe_max_empty_dest)
      then have {tas  $\in$  ts_tuple_rel (set (linearize data_in')).
        valid_tuple tuple_since tas  $\wedge$  as = snd tas} = {}
        using ts_tuple_rel_mono[OF set_lin_data_in'] by auto
      then show ?thesis
        unfolding tuple_in_None_None[OF None] using iffD2[OF safe_max_empty, symmetric] by
blast
  next
  case (Some t)
  show ?thesis
  proof (cases  $\exists X. (t, X) \in \text{set discard} \wedge as \in X$ )
    case True
      then obtain X where X_def:  $(t, X) \in \text{set discard}$  as  $as \in X$ 
        by auto
      have enat  $(nt - t) > \text{right } I$ 
        using X_def(1) unfolding discard_alt by simp
      moreover have  $\bigwedge t'. (t', as) \in \text{ts\_tuple\_rel (set (linearize data\_in'))} \implies$ 
        valid_tuple tuple_since  $(t', as) \implies t' \leq t$ 
        using valid_before Some safe_max_Some_dest_le[OF finite_fst_ts_tuple_rel]
        by (fastforce simp add: image_iff)
      ultimately have {tas  $\in$  ts_tuple_rel (set (linearize data_in')).
        valid_tuple tuple_since tas  $\wedge$  as = snd tas} = {}
        unfolding lin_data_in' using ts_tuple_rel_set_filter
        by (auto simp add: ts_tuple_rel_def)
  end

```

```

    (meson diff le_mono2 enat_ord_simps(2) leD le_less_trans)
  then show ?thesis
    unfolding tuple_in_Some_None[OF Some X_def(2,1)]
    using iffD2[OF safe_max_empty, symmetric] by blast
next
case False
then have lookup_Some: Mapping.lookup tuple_in' as = Some t
  using tuple_in_Some_Some[OF Some] by auto
have t_as: (t, as) ∈ ts_tuple_rel (set (linearize data_in))
  valid_tuple tuple_since (t, as)
  using valid_before_Some by (auto dest: safe_max_Some_dest_in[OF finite_fst_ts_tuple_rel])
then obtain X where X_def: as ∈ X (t, X) ∈ set (linearize data_in)
  by (auto simp add: ts_tuple_rel_def)
have (t, X) ∈ set (linearize data_in')
  using X_def False unfolding discard_alt lin_data_in' by auto
then have t_in_fst: t ∈ fst ' {tas ∈ ts_tuple_rel (set (linearize data_in'))}.
  valid_tuple tuple_since tas ∧ as = snd tas}
  using t_as(2) X_def(1) by (auto simp add: ts_tuple_rel_def image_iff)
have ∧t'. (t', as) ∈ ts_tuple_rel (set (linearize data_in')) ⇒
  valid_tuple tuple_since (t', as) ⇒ t' ≤ t
  using valid_before_Some safe_max_Some_dest_le[OF finite_fst_ts_tuple_rel]
  by (fastforce simp add: image_iff)
then have Max (fst ' {tas ∈ ts_tuple_rel (set (linearize data_in'))}.
  valid_tuple tuple_since tas ∧ as = snd tas}) = t
  using Max_eqI[OF finite_fst_ts_tuple_rel, OF _ t_in_fst]
  ts_tuple_rel_mono[OF set_lin_data_in'] by fastforce
then show ?thesis
  unfolding lookup_Some using t_in_fst by (auto simp add: safe_max_def)
qed
qed
qed
have table_in: table (args_n args) (args_R args) (Mapping.keys tuple_in')
  using tuple_in'_keys valid_before by (auto simp add: table_def)
have ts_tuple_rel (set auxlist) =
  {as ∈ ts_tuple_rel (set (linearize data_prev)) ∪ set (linearize data_in)}.
  valid_tuple tuple_since as}
  using valid_before by auto
then have ts_tuple_rel (set (filter (λ(t, rel). enat (nt - t) ≤ right I) auxlist)) =
  {as ∈ ts_tuple_rel (set (linearize data_prev')) ∪ set (linearize data_in')}.
  valid_tuple tuple_since as}
  unfolding lin_data_prev' lin_data_in' ts_tuple_rel_Un ts_tuple_rel_filter by auto
then show ?thesis
  using data_prev'_def data_in'_def tuple_in'_def discard_def valid_before nt_mono
  sorted_lin_data_prev' sorted_lin_data_in' lin_data_prev' lin_data_in' lookup_tuple_in'
  table_in unfolding I_def
  by (auto simp only: valid_mmsaux_simps shift_end_simps Let_def split: prod.splits) auto
qed

lemma valid_shift_end_mmsaux: valid_mmsaux args cur aux auxlist ⇒ nt ≥ cur ⇒
  valid_mmsaux args cur (shift_end args nt aux)
  (filter (λ(t, rel). enat (nt - t) ≤ right (args_ivl args)) auxlist)
  using valid_shift_end_mmsaux_unfolded by (cases aux) fast

setup_lifting type_definition_mapping

lift_definition upd_set :: ('a, 'b) mapping ⇒ ('a ⇒ 'b) ⇒ 'a set ⇒ ('a, 'b) mapping is
  λm f X a. if a ∈ X then Some (f a) else m a .

```

lemma *Mapping_lookup_upd_set*: $Mapping.lookup (upd_set\ m\ f\ X)\ a =$
 (if $a \in X$ then $Some\ (f\ a)$ else $Mapping.lookup\ m\ a$)
by (*simp add*: $Mapping.lookup.rep_eq\ upd_set.rep_eq$)

lemma *Mapping_upd_set_keys*: $Mapping.keys (upd_set\ m\ f\ X) = Mapping.keys\ m \cup X$
by (*auto simp add*: $Mapping_lookup_upd_set\ dest!$: $Mapping_keys_dest$ *intro*: $Mapping_keys_intro$)

lift_definition *upd_keys_on* :: $('a, 'b)\ mapping \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow$
 $('a, 'b)\ mapping$ **is**
 $\lambda m\ f\ X\ a.\ case\ Mapping.lookup\ m\ a\ of\ Some\ b \Rightarrow Some\ (if\ a \in X\ then\ f\ a\ b\ else\ b)$
 $| None \Rightarrow None$.

lemma *Mapping_lookup_upd_keys_on*: $Mapping.lookup (upd_keys_on\ m\ f\ X)\ a =$
 (case $Mapping.lookup\ m\ a$ of $Some\ b \Rightarrow Some\ (if\ a \in X\ then\ f\ a\ b\ else\ b)$ | $None \Rightarrow None$)
by (*simp add*: $Mapping.lookup.rep_eq\ upd_keys_on.rep_eq$)

lemma *Mapping_upd_keys_sub*: $Mapping.keys (upd_keys_on\ m\ f\ X) = Mapping.keys\ m$
by (*auto simp add*: $Mapping_lookup_upd_keys_on\ dest!$: $Mapping_keys_dest$ *intro*: $Mapping_keys_intro$
split: $option.splits$)

lemma *fold_append_queue_rep*: $linearize (fold (\lambda x\ q.\ append_queue\ x\ q)\ xs\ q) = linearize\ q\ @\ xs$
by (*induction xs arbitrary*: q) (*auto simp add*: $append_queue_rep$)

lemma *Max_Un_absorb*:
assumes *finite X* $X \neq \{\}$ *finite Y* $(\bigwedge y.\ y \in Y \Longrightarrow x \in X \Longrightarrow y \leq x)$
shows $Max\ (X \cup Y) = Max\ X$

proof –

have $Max\ X\ in\ X$: $Max\ X \in X$
using $Max_in[OF\ assms(1,2)]$.
have $Max\ X\ in\ XY$: $Max\ X \in X \cup Y$
using $Max_in[OF\ assms(1,2)]$ **by auto**
have *fin*: *finite* $(X \cup Y)$
using $assms(1,3)$ **by auto**
have $Y_le_Max\ X$: $\bigwedge y.\ y \in Y \Longrightarrow y \leq Max\ X$
using $assms(4)[OF_ Max_X_in_X]$.
have $XY_le_Max\ X$: $\bigwedge y.\ y \in X \cup Y \Longrightarrow y \leq Max\ X$
using $Max_ge[OF\ assms(1)]\ Y_le_Max_X$ **by auto**
show *?thesis*
using $Max_eqI[OF\ fin\ XY_le_Max_X\ Max_X_in_XY]$ **by auto**

qed

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 (*induction xs arbitrary*: m)

case *Nil*

then show *?case* **by simp**

next

case $(Cons\ x\ xs)$

obtain $x1\ x2$ **where** $x = (x1, x2)$ **by** (*cases x*)

have $Mapping.lookup (fold (\lambda(t, X)\ m.\ upd_set\ m\ (\lambda_.\ t)\ (Z\ X\ t))\ xs\ (upd_set\ m\ (\lambda_.\ x1)\ (Z\ x2\ x1)))$

as =

$Mapping.lookup (upd_set\ m\ (\lambda_.\ x1)\ (Z\ x2\ x1))\ as$

using *Cons* **by auto**

also have $Mapping.lookup (upd_set\ m\ (\lambda_.\ x1)\ (Z\ x2\ x1))\ as = Mapping.lookup\ m\ as$

using *Cons.prem*s **by** (*auto simp*: $\langle x = (x1, x2) \rangle Mapping_lookup_upd_set$)

finally show *?case* **by** (*simp add*: $\langle x = (x1, x2) \rangle$)

qed

lemma *Mapping_lookup_fold_upd_set_max*: $\{(t, X) \in \text{set } xs. as \in Z X t\} \neq \{\} \implies$
sorted (*map fst xs*) \implies
Mapping.lookup (*fold* ($\lambda(t, X) m. \text{upd_set } m (\lambda_. t) (Z X t)$) *xs m*) *as* =
Some (*Max* (*fst* ' $\{(t, X) \in \text{set } xs. as \in Z X t\}$))
proof (*induction xs arbitrary: m*)
case (*Cons x xs*)
obtain *t X where tX_def: x = (t, X)*
by (*cases x auto*)
have *set_fst_eq: (fst ' $\{(t, X). (t, X) \in \text{set } (x \# xs) \wedge as \in Z X t\}$) =*
((fst ' $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\}$) \cup
(if $as \in Z X t$ then $\{t\}$ else $\{\}$))
using *image_iff by (fastforce simp add: tX_def split: if_splits)*
show *?case*
proof (*cases $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\} \neq \{\}$*)
case *True*
have $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\} \subseteq \text{set } xs$
by *auto*
then have *fin: finite (fst ' $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\}$)*
by (*simp add: finite_subset*)
have *Max (insert t (fst ' $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\}$)) =*
Max (fst ' $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\}$)
using *Max_Un_absorb[OF fin, of $\{t\}$] True Cons(3) tX_def by auto*
then show *?thesis*
using *Cons True unfolding set_fst_eq by auto*
next
case *False*
then have *empty: $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\} = \{\}$*
by *auto*
then have $as \in Z X t$
using *Cons(2) set_fst_eq by fastforce*
then show *?thesis*
using *Mapping_lookup_fold_upd_set_idle[OF empty] unfolding set_fst_eq empty*
by (*auto simp add: Mapping_lookup_upd_set tX_def*)
qed
qed simp

fun *add_new_ts_mmsaux'* :: $args \Rightarrow ts \Rightarrow 'a \text{ mmsaux} \Rightarrow 'a \text{ mmsaux}$ **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 ($\lambda(t, X). nt - t \geq \text{left } I$) data_prev;
data_in = fold ($\lambda(t, X) data_in. \text{append_queue } (t, X) data_in$) move data_in;
tuple_in = fold ($\lambda(t, X) tuple_in. \text{upd_set } tuple_in (\lambda_. t)$
{ $as \in X. \text{valid_tuple } tuple_since (t, as)$ } move tuple_in in
(nt, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))

lemma *Mapping_keys_fold_upd_set*: $k \in \text{Mapping.keys} (\text{fold } (\lambda(t, X) m. \text{upd_set } m (\lambda_. t) (Z t X))$
xs m) $\implies k \in \text{Mapping.keys } m \vee (\exists (t, X) \in \text{set } xs. k \in Z t X)$
by (*induction xs arbitrary: m*) (*fastforce simp add: Mapping_upd_set_keys*)**+**

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 –
define *I where I = args_ivl args*

```

define n where n = args_n args
define L where L = args_L args
define R where R = args_R args
define pos where pos = args_pos args
define data_prev' where data_prev'  $\equiv$  dropWhile_queue ( $\lambda(t, X). nt - t \geq \text{left } I$ ) data_prev
define move where move  $\equiv$  takeWhile ( $\lambda(t, X). nt - t \geq \text{left } I$ ) (linearize data_prev)
define data_in' where data_in'  $\equiv$  fold ( $\lambda(t, X)$  data_in. append_queue (t, X) data_in)
  move data_in
define tuple_in' where tuple_in'  $\equiv$  fold ( $\lambda(t, X)$  tuple_in. upd_set tuple_in ( $\lambda_. t$ )
  {as  $\in$  X. valid_tuple tuple_since (t, as)}) move tuple_in
have tuple_in'_keys:  $\bigwedge as. as \in \text{Mapping.keys } tuple\_in' \implies as \in \text{Mapping.keys } tuple\_in \vee$ 
 $(\exists (t, X) \in \text{set } move. as \in \{as \in X. \text{valid\_tuple } tuple\_since (t, as)\})$ 
  using Mapping_keys_fold_upd_set[of  $\_ \lambda t X. \{as \in X. \text{valid\_tuple } tuple\_since (t, as)\}$ ]
  by (auto simp add: tuple_in'_def)
have F1: sorted (map fst (linearize data_in))  $\forall t \in \text{fst ' set } (linearize data\_in). t \leq nt$ 
 $\forall t \in \text{fst ' set } (linearize data\_in). t \leq ot \wedge ot - t \geq \text{left } I$ 
  using valid_before nt_mono unfolding I_def by auto
have F2: sorted (map fst (linearize data_prev))  $\forall t \in \text{fst ' set } (linearize data\_prev). t \leq nt$ 
 $\forall t \in \text{fst ' set } (linearize data\_prev). t \leq ot \wedge ot - t < \text{left } I$ 
  using valid_before nt_mono unfolding I_def by auto
have lin_data_prev': linearize data_prev' =
  filter ( $\lambda(t, X). nt - t < \text{left } I$ ) (linearize data_prev)
  unfolding data_prev'_def dropWhile_queue_rep dropWhile_filter'[OF F2(1,2)] ..
have move_filter: move = filter ( $\lambda(t, X). nt - t \geq \text{left } I$ ) (linearize data_prev)
  unfolding move_def takeWhile_filter'[OF F2(1,2)] ..
then have sorted_move: sorted (map fst move)
  using sorted_filter F2 by auto
have  $\forall t \in \text{fst ' set } move. t \leq ot \wedge ot - t < \text{left } I$ 
  using move_filter F2(3) set_filter by auto
then have fst_set_before:  $\forall t \in \text{fst ' set } (linearize data\_in). \forall t' \in \text{fst ' set } move. t \leq t'$ 
  using F1(3) by fastforce
then have fst_ts_tuple_rel_before:  $\forall t \in \text{fst ' ts\_tuple\_rel } (set (linearize data\_in)).$ 
 $\forall t' \in \text{fst ' ts\_tuple\_rel } (set move). t \leq t'$ 
  by (fastforce simp add: ts_tuple_rel_def)
have sorted_lin_data_prev': sorted (map fst (linearize data_prev'))
  unfolding lin_data_prev' using sorted_filter F2 by auto
have lin_data_in': linearize data_in' = linearize data_in @ move
  unfolding data_in'_def using fold_append_queue_rep by fastforce
have sorted_lin_data_in': sorted (map fst (linearize data_in'))
  unfolding lin_data_in' using F1(1) sorted_move fst_set_before by (simp add: sorted_append)
have set_lin_prev'_in': set (linearize data_prev')  $\cup$  set (linearize data_in) =
set (linearize data_prev)  $\cup$  set (linearize data_in)
  using lin_data_prev' lin_data_in' move_filter by auto
have ts_tuple_rel': ts_tuple_rel (set auxlist) =
{tas  $\in$  ts_tuple_rel (set (linearize data_prev')  $\cup$  set (linearize data_in)).
valid_tuple tuple_since tas}
  unfolding set_lin_prev'_in' using valid_before by auto
have lookup':  $\bigwedge as. \text{Mapping.lookup } tuple\_in' as = \text{safe\_max } (fst ' \{tas \in ts\_tuple\_rel (set (linearize data\_in')).$ 
valid_tuple tuple\_since tas  $\wedge as = \text{snd } tas\})$ 
proof –
  fix as
  show Mapping.lookup tuple_in' as = safe_max (fst ' \{tas \in ts\_tuple\_rel (set (linearize data\_in')).
valid_tuple tuple\_since tas  $\wedge as = \text{snd } tas\}$ )
  proof (cases  $\{(t, X) \in \text{set } move. as \in X \wedge \text{valid\_tuple } tuple\_since (t, as)\} = \{\}$ )
  case True
  have move_absorb:  $\{tas \in ts\_tuple\_rel (set (linearize data\_in)).$ 

```

```

    valid_tuple tuple_since tas ∧ as = snd tas} =
    {tas ∈ ts_tuple_rel (set (linearize data_in @ move)).
    valid_tuple tuple_since tas ∧ as = snd tas}
  using True by (auto simp add: ts_tuple_rel_def)
  have Mapping.lookup tuple_in as =
    safe_max (fst ' {tas ∈ ts_tuple_rel (set (linearize data_in)).
    valid_tuple tuple_since tas ∧ as = snd tas})
  using valid_before by auto
  then have Mapping.lookup tuple_in as =
    safe_max (fst ' {tas ∈ ts_tuple_rel (set (linearize data_in')).
    valid_tuple tuple_since tas ∧ as = snd tas})
  unfolding lin_data_in' move_absorb .
  then show ?thesis
  using Mapping_lookup_fold_upd_set_idle[of move as
    λX t. {as ∈ X. valid_tuple tuple_since (t, as)}] True
  unfolding tuple_in'_def by auto
next
case False
  have split: fst ' {tas ∈ ts_tuple_rel (set (linearize data_in'))}.
    valid_tuple tuple_since tas ∧ as = snd tas} =
    fst ' {tas ∈ ts_tuple_rel (set move). valid_tuple tuple_since tas ∧ as = snd tas} ∪
    fst ' {tas ∈ ts_tuple_rel (set (linearize data_in)).
    valid_tuple tuple_since tas ∧ as = snd tas}
  unfolding lin_data_in' set_append ts_tuple_rel_Un by auto
  have max_eq: Max (fst ' {tas ∈ ts_tuple_rel (set move).
    valid_tuple tuple_since tas ∧ as = snd tas}) =
    Max (fst ' {tas ∈ ts_tuple_rel (set (linearize data_in')).
    valid_tuple tuple_since tas ∧ as = snd tas})
  unfolding split using False fst_ts_tuple_rel_before
  by (fastforce simp add: ts_tuple_rel_def
    intro!: Max_Un_absorb[OF finite_fst_ts_tuple_rel_finite_fst_ts_tuple_rel_symmetric])
  have fst ' {(t, X). (t, X) ∈ set move ∧ as ∈ {as ∈ X. valid_tuple tuple_since (t, as)}} =
    fst ' {tas ∈ ts_tuple_rel (set move). valid_tuple tuple_since tas ∧ as = snd tas}
  by (auto simp add: ts_tuple_rel_def image_iff)
  then have Mapping.lookup tuple_in' as = Some (Max (fst ' {tas ∈ ts_tuple_rel (set move).
    valid_tuple tuple_since tas ∧ as = snd tas}))
  using Mapping_lookup_fold_upd_set_max[of move as
    λX t. {as ∈ X. valid_tuple tuple_since (t, as)}, OF_sorted_move] False
  unfolding tuple_in'_def by (auto simp add: ts_tuple_rel_def)
  then show ?thesis
  unfolding max_eq using False
  by (auto simp add: safe_max_def lin_data_in' ts_tuple_rel_def)
qed
qed
have table_in': table n R (Mapping.keys tuple_in')
proof -
{
  fix as
  assume assm: as ∈ Mapping.keys tuple_in'
  have wf_tuple n R as
  using tuple_in'_keys[OF assm]
  proof (rule disjE)
  assume as ∈ Mapping.keys tuple_in
  then show wf_tuple n R as
  using valid_before by (auto simp add: table_def n_def R_def)
  next
  assume ∃(t, X) ∈ set move. as ∈ {as ∈ X. valid_tuple tuple_since (t, as)}
  then obtain t X where tX_def: (t, X) ∈ set move as ∈ X

```

```

    by auto
  then have as ∈ ∪ (snd ' set (linearize data_prev))
    unfolding move_def using set_takeWhileD by force
  then show wf_tuple n R as
    using valid_before by (auto simp add: n_def R_def)
  qed
}
then show ?thesis
  by (auto simp add: table_def)
qed
have data_prev'_move: (data_prev', move) =
  takedownWhile_queue (λ(t, X). nt - t ≥ left I) data_prev
  using takedownWhile_queuefst takedownWhile_queue_snd data_prev'_def move_def
  by (metis surjective_pairing)
moreover have valid_mmsaux args nt (nt, gc, maskL, maskR, data_prev', data_in',
  tuple_in', tuple_since) auxlist
  using lin_data_prev' sorted_lin_data_prev' lin_data_in' move_filter sorted_lin_data_in'
  nt_mono valid_before ts_tuple_rel' lookup' table_in' unfolding I_def
  by (auto simp only: valid_mmsaux.simps Let_def n_def R_def split: option.splits) auto

ultimately show ?thesis
  by (auto simp only: add_new_ts_mmsaux'.simps Let_def data_in'_def tuple_in'_def I_def
  split: prod.splits)
qed

lemma valid_add_new_ts_mmsaux': valid_mmsaux args cur aux auxlist ⇒ nt ≥ cur ⇒
  valid_mmsaux args nt (add_new_ts_mmsaux' args nt aux) auxlist
  using valid_add_new_ts_mmsaux'_unfolded by (cases aux) fast

definition add_new_ts_mmsaux :: args ⇒ ts ⇒ 'a mmsaux ⇒ '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 ≥ cur
  shows valid_mmsaux args nt (add_new_ts_mmsaux args nt aux)
    (filter (λ(t, rel). enat (nt - t) ≤ right (args_ivl args)) auxlist)
  using valid_add_new_ts_mmsaux'[OF valid_shift_end_mmsaux[OF assms] assms(2)]
  unfolding add_new_ts_mmsaux_def .

fun join_mmsaux :: args ⇒ 'a table ⇒ 'a mmsaux ⇒ '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 (∀ i ∈ set maskL. ¬i) then
      (let nones = replicate (length maskL) None;
       take_all = (pos ↔ nones ∈ 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 (λas _. proj_tuple_in_join pos maskL as X) tuple_in;
       tuple_since = Mapping.filter (λas _. proj_tuple_in_join pos maskL as X) tuple_since in
      (t, gc, maskL, maskR, data_prev, data_in, tuple_in, tuple_since))))

fun join_mmsaux_abs :: args ⇒ 'a table ⇒ 'a mmsaux ⇒ '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 (λas _. proj_tuple_in_join pos maskL as X) tuple_in;
    tuple_since = Mapping.filter (λas _. proj_tuple_in_join pos maskL as 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 $\text{Mapping.filter } f m = \text{Mapping.filter } f' m$

proof –

have $\bigwedge k. \text{Mapping.lookup } (\text{Mapping.filter } f m) k = \text{Mapping.lookup } (\text{Mapping.filter } f' m) k$
using *cong*

by (*fastforce simp add: Mapping.lookup_filter intro: Mapping_keys_intro split: option.splits*)

then show *?thesis*

by (*simp add: mapping_eqI*)

qed

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 $\text{join_mmsaux_abs } \text{args } X (\text{nt}, \text{gc}, \text{maskL}, \text{maskR}, \text{data_prev}, \text{data_in}, \text{tuple_in}, \text{tuple_since}) =$
 $\text{join_mmsaux_abs } \text{args } X (\text{nt}, \text{gc}, \text{maskL}, \text{maskR}, \text{data_prev}, \text{data_in}, \text{tuple_in}, \text{tuple_since})$

proof (*cases maskL = maskR*)

case *True*

define *n* **where** *n* = args_n args

define *L* **where** *L* = args_L args

define *pos* **where** *pos* = args_pos args

have *keys_wf_in*: $\bigwedge as. as \in \text{Mapping.keys } \text{tuple_in} \implies \text{wf_tuple } n L as$

using *wf_tuple_change_base valid_before True* **by** (*fastforce simp add: table_def n_def L_def*)

have *cong_in*: $\bigwedge as n. as \in \text{Mapping.keys } \text{tuple_in} \implies$

proj_tuple_in_join pos maskL as X \longleftrightarrow *join_cond pos X as*

using *proj_tuple_in_join_mask_idle[OF keys_wf_in] valid_before*

by (*auto simp only: valid_mmsaux.simps n_def L_def pos_def*)

have *keys_wf_since*: $\bigwedge as. as \in \text{Mapping.keys } \text{tuple_since} \implies \text{wf_tuple } n L as$

using *wf_tuple_change_base valid_before True* **by** (*fastforce simp add: table_def n_def L_def*)

have *cong_since*: $\bigwedge as n. as \in \text{Mapping.keys } \text{tuple_since} \implies$

proj_tuple_in_join pos maskL as X \longleftrightarrow *join_cond pos X as*

using *proj_tuple_in_join_mask_idle[OF keys_wf_since] valid_before*

by (*auto simp only: valid_mmsaux.simps n_def L_def pos_def*)

show *?thesis*

using *True Mapping_filter_cong[OF cong_in, of tuple_in λk _. k]*

Mapping_filter_cong[OF cong_since, of tuple_since λk _. k]

by (*auto simp add: pos_def*)

next

case *False*

define *n* **where** *n* = args_n args

define *L* **where** *L* = args_L args

define *R* **where** *R* = args_R args

define *pos* **where** *pos* = args_pos args

from *False* **show** *?thesis*

proof (*cases* $\forall i \in \text{set } \text{maskL}. \neg i$)

case *True*

have *length_maskL*: *length maskL* = *n*

using *valid_before* **by** (*auto simp add: join_mask_def n_def*)

have *proj_rep*: $\bigwedge as. \text{wf_tuple } n R as \implies \text{proj_tuple } \text{maskL } as = \text{replicate } (\text{length } \text{maskL}) \text{ None}$

using *True proj_tuple_replicate* **by** (*force simp add: length_maskL wf_tuple_def*)

have *keys_wf_in*: $\bigwedge as. as \in \text{Mapping.keys } \text{tuple_in} \implies \text{wf_tuple } n R as$

```

using valid_before by (auto simp add: table_def n_def R_def)
have keys_wf_since:  $\bigwedge as. as \in Mapping.keys\ tuple\_since \implies wf\_tuple\ n\ R\ as$ 
using valid_before by (auto simp add: table_def n_def R_def)
have  $\bigwedge as. Mapping.lookup\ (Mapping.filter\ (\lambda as\ \_.\ proj\_tuple\_in\_join\ pos\ maskL\ as\ X)\$ 
tuple_in) as = Mapping.lookup (if (pos  $\longleftrightarrow$  replicate (length maskL) None  $\in$  X)
then tuple_in else Mapping.empty) as
using proj_rep[OF keys_wf_in]
by (auto simp add: Mapping.lookup_filter Mapping.lookup_empty proj_tuple_in_join_def
Mapping_keys_intro split: option.splits)
moreover have  $\bigwedge as. Mapping.lookup\ (Mapping.filter\ (\lambda as\ \_.\ proj\_tuple\_in\_join\ pos\ maskL\ as\ X)\$ 
tuple_since) as = Mapping.lookup (if (pos  $\longleftrightarrow$  replicate (length maskL) None  $\in$  X)
then tuple_since else Mapping.empty) as
using proj_rep[OF keys_wf_since]
by (auto simp add: Mapping.lookup_filter Mapping.lookup_empty proj_tuple_in_join_def
Mapping_keys_intro split: option.splits)
ultimately show ?thesis
using False True by (auto simp add: mapping_eqI Let_def pos_def)
qed (auto simp add: Let_def)
qed

```

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 ( $\lambda(t, rel). (t, join\ rel\ (args\_pos\ args)\ X)$ ) auxlist)

```

proof –

```

define n where n = args_n args
define L where L = args_L args
define R where R = args_R args
define pos where pos = args_pos args
note table_left = table_left'[unfolded n_def[symmetric] L_def[symmetric]]
define tuple_in' where tuple_in'  $\equiv$ 
Mapping.filter ( $\lambda as\ \_.\ proj\_tuple\_in\_join\ pos\ maskL\ as\ X$ ) tuple_in
define tuple_since' where tuple_since'  $\equiv$ 
Mapping.filter ( $\lambda as\ \_.\ proj\_tuple\_in\_join\ pos\ maskL\ as\ X$ ) tuple_since
have tuple_in_None_None:  $\bigwedge as. Mapping.lookup\ tuple\_in\ as = None \implies$ 
Mapping.lookup tuple_in' as = None
unfolding tuple_in'_def using Mapping_lookup_filter_not_None by fastforce
have tuple_in'_keys:  $\bigwedge as. as \in Mapping.keys\ tuple\_in' \implies as \in Mapping.keys\ tuple\_in$ 
using tuple_in_None_None
by (fastforce intro: Mapping_keys_intro dest: Mapping_keys_dest)
have tuple_since_None_None:  $\bigwedge as. Mapping.lookup\ tuple\_since\ as = None \implies$ 
Mapping.lookup tuple_since' as = None
unfolding tuple_since'_def using Mapping_lookup_filter_not_None by fastforce
have tuple_since'_keys:  $\bigwedge as. as \in Mapping.keys\ tuple\_since' \implies as \in Mapping.keys\ tuple\_since$ 
using tuple_since_None_None
by (fastforce intro: Mapping_keys_intro dest: Mapping_keys_dest)
have ts_tuple_rel': ts_tuple_rel (set (map ( $\lambda(t, rel). (t, join\ rel\ pos\ X)$ ) auxlist)) =
{tas  $\in$  ts_tuple_rel (set (linearize data_prev)  $\cup$  set (linearize data_in)).
valid_tuple tuple_since' tas}
proof (rule set_eqI, rule iffI)
fix tas
assume assm: tas  $\in$  ts_tuple_rel (set (map ( $\lambda(t, rel). (t, join\ rel\ pos\ X)$ ) auxlist))
then obtain t as Z where tas_def: tas = (t, as) as  $\in$  join Z pos X (t, Z)  $\in$  set auxlist
(t, join Z pos X)  $\in$  set (map ( $\lambda(t, rel). (t, join\ rel\ pos\ X)$ ) auxlist)
by (fastforce simp add: ts_tuple_rel_def)

```

```

from tas_def(3) have table_Z: table n R Z
  using valid_before by (auto simp add: n_def R_def)
have proj: as ∈ Z proj_tuple_in_join pos maskL as X
  using tas_def(2) join_sub[OF _ table_left table_Z] valid_before
  by (auto simp add: n_def L_def R_def pos_def)
then have (t, as) ∈ ts_tuple_rel (set (auxlist))
  using tas_def(3) by (auto simp add: ts_tuple_rel_def)
then have tas_in: (t, as) ∈ ts_tuple_rel
  (set (linearize data_prev) ∪ set (linearize data_in)) valid_tuple tuple_since (t, as)
  using valid_before by auto
then obtain t' where t'_def: Mapping.lookup tuple_since as = Some t' t ≥ t'
  by (auto simp add: valid_tuple_def split: option.splits)
then have valid_tuple_since': valid_tuple tuple_since' (t, as)
  using proj(2)
  by (auto simp add: tuple_since'_def Mapping_lookup_filter_Some valid_tuple_def)
show tas ∈ {tas ∈ ts_tuple_rel (set (linearize data_prev) ∪ set (linearize data_in)).
  valid_tuple tuple_since' tas}
  using tas_in valid_tuple_since' unfolding tas_def(1)[symmetric] by auto
next
fix tas
assume assm: tas ∈ {tas ∈ ts_tuple_rel
  (set (linearize data_prev) ∪ set (linearize data_in)). valid_tuple tuple_since' tas}
then obtain t as where tas_def: tas = (t, as) valid_tuple tuple_since' (t, as)
  by (auto simp add: ts_tuple_rel_def)
from tas_def(2) have valid_tuple tuple_since (t, as)
  unfolding tuple_since'_def using Mapping_lookup_filter_not_None
  by (force simp add: valid_tuple_def split: option.splits)
then have (t, as) ∈ ts_tuple_rel (set auxlist)
  using valid_before assm tas_def(1) by auto
then obtain Z where Z_def: as ∈ Z (t, Z) ∈ set auxlist
  by (auto simp add: ts_tuple_rel_def)
then have table_Z: table n R Z
  using valid_before by (auto simp add: n_def R_def)
from tas_def(2) have proj_tuple_in_join pos maskL as X
  unfolding tuple_since'_def using Mapping_lookup_filter_Some_P
  by (fastforce simp add: valid_tuple_def split: option.splits)
then have as_in_join: as ∈ join Z pos X
  using join_sub[OF _ table_left table_Z] Z_def(1) valid_before
  by (auto simp add: n_def L_def R_def pos_def)
then show tas ∈ ts_tuple_rel (set (map (λ(t, rel). (t, join rel pos X)) auxlist))
  using Z_def unfolding tas_def(1) by (auto simp add: ts_tuple_rel_def)
qed
have lookup_tuple_in':  $\bigwedge as. Mapping.lookup tuple_in' as = safe\_max (fst \text{'}$ 
  {tas ∈ ts_tuple_rel (set (linearize data_in)). valid_tuple tuple_since' tas ∧ as = snd tas})
proof –
  fix as
show Mapping.lookup tuple_in' as = safe_max (fst \text{'}
  {tas ∈ ts_tuple_rel (set (linearize data_in)). valid_tuple tuple_since' tas ∧ as = snd tas})
proof (cases Mapping.lookup tuple_in as)
  case None
  then have {tas ∈ ts_tuple_rel (set (linearize data_in)).
  valid_tuple tuple_since tas ∧ as = snd tas} = {}
  using valid_before by (auto dest!: safe_max_empty_dest)
  then have {tas ∈ ts_tuple_rel (set (linearize data_in)).
  valid_tuple tuple_since' tas ∧ as = snd tas} = {}
  using Mapping_lookup_filter_not_None
  by (fastforce simp add: valid_tuple_def tuple_since'_def split: option.splits)
then show ?thesis

```

```

      unfolding tuple_in_None_None[OF None] using iffD2[OF safe_max_empty, symmetric] by
blast
next
case (Some t)
show ?thesis
proof (cases proj_tuple_in_join pos maskL as X)
case True
then have lookup_tuple_in': Mapping.lookup tuple_in' as = Some t
  using Some unfolding tuple_in'_def by (simp add: Mapping_lookup_filter_Some)
have (t, as) ∈ ts_tuple_rel (set (linearize data_in)) valid_tuple tuple_since (t, as)
  using valid_before Some by (auto dest: safe_max_Some_dest_in[OF finite_fst_ts_tuple_rel])
then have t_in_fst: t ∈ fst ' {tas ∈ ts_tuple_rel (set (linearize data_in)).
  valid_tuple tuple_since' tas ∧ as = snd tas}
  using True by (auto simp add: image_iff valid_tuple_def tuple_since'_def
  Mapping_lookup_filter_Some split: option.splits)
have ∧t'. valid_tuple tuple_since' (t', as) ⇒ valid_tuple tuple_since (t', as)
  using Mapping_lookup_filter_not_None
  by (fastforce simp add: valid_tuple_def tuple_since'_def split: option.splits)
then have ∧t'. (t', as) ∈ ts_tuple_rel (set (linearize data_in)) ⇒
  valid_tuple tuple_since' (t', as) ⇒ t' ≤ t
  using valid_before Some safe_max_Some_dest_le[OF finite_fst_ts_tuple_rel]
  by (fastforce simp add: image_iff)
then have Max (fst ' {tas ∈ ts_tuple_rel (set (linearize data_in)).
  valid_tuple tuple_since' tas ∧ as = snd tas}) = t
  using Max_eqI[OF finite_fst_ts_tuple_rel[of linearize data_in],
  OF_t_in_fst] by fastforce
then show ?thesis
  unfolding lookup_tuple_in' using t_in_fst by (auto simp add: safe_max_def)
next
case False
then have lookup_tuple': Mapping.lookup tuple_in' as = None
  Mapping.lookup tuple_since' as = None
  unfolding tuple_in'_def tuple_since'_def
  by (auto simp add: Mapping_lookup_filter_None)
then have ∧tas. ¬(valid_tuple tuple_since' tas ∧ as = snd tas)
  by (auto simp add: valid_tuple_def split: option.splits)
then show ?thesis
  unfolding lookup_tuple' by (auto simp add: safe_max_def)
qed
qed
qed
have table_join': ∧t ys. (t, ys) ∈ set auxlist ⇒ table n R (join ys pos X)
proof -
fix t ys
assume (t, ys) ∈ set auxlist
then have table_ys: table n R ys
  using valid_before
  by (auto simp add: n_def L_def R_def pos_def)
show table n R (join ys pos X)
  using join_table[OF table_ys table_left, of pos R] valid_before
  by (auto simp add: n_def L_def R_def pos_def)
qed
have table_in': table n R (Mapping.keys tuple_in')
  using tuple_in'_keys valid_before
  by (auto simp add: n_def L_def R_def pos_def table_def)
have table_since': table n R (Mapping.keys tuple_since')
  using tuple_since'_keys valid_before
  by (auto simp add: n_def L_def R_def pos_def table_def)

```

```

show ?thesis
  unfolding join_mmsaux_abs_eq[OF valid_before table_left']
  using valid_before ts_tuple_rel' lookup_tuple_in' tuple_in'_def tuple_since'_def table_join'
    Mapping_filter_keys[of tuple_since  $\lambda$ as. case as of Some  $t \Rightarrow t \leq nt$ ]
    table_in' table_since' by (auto simp add: n_def L_def R_def pos_def table_def Let_def)
qed

```

```

lemma valid_join_mmsaux: valid_mmsaux args cur aux auxlist  $\Rightarrow$ 
  table (args_n args) (args_L args) X  $\Rightarrow$  valid_mmsaux args cur
  (join_mmsaux args X aux) (map ( $\lambda$ (t, rel). (t, join_rel (args_pos args) X)) auxlist)
using valid_join_mmsaux_unfolded by (cases aux) fast

```

```

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 –

```

define n where n = args_n args
define L where L = args_L args
define R where R = args_R args
define pos where pos = args_pos args
define all_tuples where all_tuples  $\equiv$   $\bigcup$  (snd ' (set (linearize data_prev)  $\cup$ 
  set (linearize data_in)))
define tuple_since' where tuple_since'  $\equiv$  Mapping.filter ( $\lambda$ as _. as  $\in$  all_tuples) tuple_since
have tuple_since_None_None:  $\bigwedge$ as. Mapping.lookup tuple_since as = None  $\Rightarrow$ 
  Mapping.lookup tuple_since' as = None
unfolding tuple_since'_def using Mapping_lookup_filter_not_None by fastforce
have tuple_since'_keys:  $\bigwedge$ as. as  $\in$  Mapping.keys tuple_since'  $\Rightarrow$  as  $\in$  Mapping.keys tuple_since
using tuple_since_None_None
by (fastforce intro: Mapping_keys_intro dest: Mapping_keys_dest)
then have table_since': table n R (Mapping.keys tuple_since')
using valid_before by (auto simp add: table_def n_def R_def)
have data_cong:  $\bigwedge$ tas. tas  $\in$  ts_tuple_rel (set (linearize data_prev)  $\cup$ 
  set (linearize data_in))  $\Rightarrow$  valid_tuple tuple_since' tas = valid_tuple tuple_since tas

```

proof –

```

fix tas
assume assm: tas  $\in$  ts_tuple_rel (set (linearize data_prev)  $\cup$ 
  set (linearize data_in))
define t where t  $\equiv$  fst tas
define as where as  $\equiv$  snd tas
have as  $\in$  all_tuples
using assm by (force simp add: as_def all_tuples_def ts_tuple_rel_def)
then have Mapping.lookup tuple_since' as = Mapping.lookup tuple_since as
by (auto simp add: tuple_since'_def Mapping_lookup_filter_split: option.splits)
then show valid_tuple tuple_since' tas = valid_tuple tuple_since tas
by (auto simp add: valid_tuple_def as_def split: option.splits) metis

```

qed

```

then have data_in_cong:  $\bigwedge$ tas. tas  $\in$  ts_tuple_rel (set (linearize data_in))  $\Rightarrow$ 
  valid_tuple tuple_since' tas = valid_tuple tuple_since tas
by (auto simp add: ts_tuple_rel_Un)
have ts_tuple_rel (set ys) =

```

```

    {tas ∈ ts_tuple_rel (set (linearize data_prev) ∪ set (linearize data_in)).
    valid_tuple tuple_since' tas}
  using data_cong valid_before by auto
  moreover have (∀ as. Mapping.lookup tuple_in as = safe_max (fst '
    {tas ∈ ts_tuple_rel (set (linearize data_in)). valid_tuple tuple_since' tas ∧ as = snd tas}))
  using valid_before by auto (meson data_in_cong)
  moreover have (∀ as ∈ Mapping.keys tuple_since'. case Mapping.lookup tuple_since' as of
    Some t ⇒ t ≤ nt)
  using Mapping.keys_filter valid_before
  by (auto simp add: tuple_since'_def Mapping.lookup_filter split: option.splits
    intro!: Mapping_keys_intro dest: Mapping_keys_dest)
  ultimately show ?thesis
  using all_tuples_def tuple_since'_def valid_before table_since'
  by (auto simp add: n_def R_def)
qed

lemma valid_gc_mmsaux: valid_mmsaux args cur aux ys ⇒ valid_mmsaux args cur (gc_mmsaux aux)
  ys
  using valid_gc_mmsaux_unfolded by (cases aux) fast

fun gc_join_mmsaux :: args ⇒ 'a table ⇒ 'a mmsaux ⇒ '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) ∨
  gc_join_mmsaux args rel1 aux = join_mmsaux args rel1 aux
  by (cases aux) (auto simp only: gc_join_mmsaux.simps split: if_splits)

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 (λ(t, rel). (t, join_rel (args_pos args) rel1)) auxlist)
  using gc_join_mmsaux_alt[of args rel1 aux]
  using valid_join_mmsaux[OF valid_gc_mmsaux[OF assms(1)] assms(2)]
    valid_join_mmsaux[OF assms]
  by auto

fun add_new_table_mmsaux :: args ⇒ 'a table ⇒ 'a mmsaux ⇒ '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 (λ_. t) (X - Mapping.keys tuple_since) in
    (if 0 ≥ left (args_ivl args) then (t, gc, maskL, maskR, data_prev, append_queue (t, X) data_in,
      upd_set tuple_in (λ_. 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 ∪ X) # ts else (cur, X) # auxlist)
  proof -
  have cur_nt: cur = nt

```

```

using valid_before by auto
define I where  $I = \text{args\_ivl } \text{args}$ 
define n where  $n = \text{args\_n } \text{args}$ 
define L where  $L = \text{args\_L } \text{args}$ 
define R where  $R = \text{args\_R } \text{args}$ 
define pos where  $\text{pos} = \text{args\_pos } \text{args}$ 
define tuple_in' where  $\text{tuple\_in}' \equiv \text{upd\_set } \text{tuple\_in } (\lambda_. \text{nt}) \text{ X}$ 
define tuple_since' where  $\text{tuple\_since}' \equiv \text{upd\_set } \text{tuple\_since } (\lambda_. \text{nt})$ 
 $(\text{X} - \text{Mapping.keys } \text{tuple\_since})$ 
define data_prev' where  $\text{data\_prev}' \equiv \text{append\_queue } (\text{nt}, \text{X}) \text{ data\_prev}$ 
define data_in' where  $\text{data\_in}' \equiv \text{append\_queue } (\text{nt}, \text{X}) \text{ data\_in}$ 
define auxlist' where  $\text{auxlist}' \equiv (\text{case } \text{auxlist} \text{ of}$ 
 $\square \Rightarrow [(\text{nt}, \text{X})]$ 
 $| ((t, y) \# \text{ts}) \Rightarrow \text{if } t = \text{nt} \text{ then } (t, y \cup \text{X}) \# \text{ts} \text{ else } (\text{nt}, \text{X}) \# \text{auxlist})$ 
have table_in':  $\text{table } n \text{ R } (\text{Mapping.keys } \text{tuple\_in}' )$ 
using table_X valid_before
by (auto simp add: table_def tuple_in'_def Mapping_upd_set_keys n_def R_def)
have table_since':  $\text{table } n \text{ R } (\text{Mapping.keys } \text{tuple\_since}' )$ 
using table_X valid_before
by (auto simp add: table_def tuple_since'_def Mapping_upd_set_keys n_def R_def)
have tuple_since'_keys:  $\text{Mapping.keys } \text{tuple\_since} \subseteq \text{Mapping.keys } \text{tuple\_since}'$ 
using Mapping_upd_set_keys by (fastforce simp add: tuple_since'_def)
have lin_data_prev':  $\text{linearize } \text{data\_prev}' = \text{linearize } \text{data\_prev} @ [(\text{nt}, \text{X})]$ 
unfolding data_prev'_def append_queue_rep ..
have wf_data_prev':  $\bigwedge \text{as. as} \in \bigcup (\text{snd } '(\text{set } (\text{linearize } \text{data\_prev}')) \Rightarrow \text{wf\_tuple } n \text{ R } \text{as}$ 
unfolding lin_data_prev' using table_X valid_before
by (auto simp add: table_def n_def R_def)
have lin_data_in':  $\text{linearize } \text{data\_in}' = \text{linearize } \text{data\_in} @ [(\text{nt}, \text{X})]$ 
unfolding data_in'_def append_queue_rep ..
have table_auxlist':  $\forall (t, X) \in \text{set } \text{auxlist}'. \text{table } n \text{ R } \text{X}$ 
using table_X table_Un valid_before
by (auto simp add: auxlist'_def n_def R_def split: list.splits if_splits)
have lookup_tuple_since':  $\forall \text{as} \in \text{Mapping.keys } \text{tuple\_since}'.$ 
 $\text{case } \text{Mapping.lookup } \text{tuple\_since}' \text{ as of } \text{Some } t \Rightarrow t \leq \text{nt}$ 
unfolding tuple_since'_def using valid_before Mapping_lookup_upd_set[of tuple_since]
by (auto dest: Mapping_keys_dest intro!: Mapping_keys_intro split: if_splits option.splits)
have ts_tuple_rel_auxlist':  $\text{ts\_tuple\_rel } (\text{set } \text{auxlist}') =$ 
 $\text{ts\_tuple\_rel } (\text{set } \text{auxlist}) \cup \text{ts\_tuple\_rel } \{(\text{nt}, \text{X})\}$ 
unfolding auxlist'_def
using ts_tuple_rel_ext ts_tuple_rel_ext' ts_tuple_rel_ext_Cons ts_tuple_rel_ext_Cons'
by (fastforce simp: ts_tuple_rel_def split: list.splits if_splits)
have valid_tuple_nt_X:  $\bigwedge \text{tas. tas} \in \text{ts\_tuple\_rel } \{(\text{nt}, \text{X})\} \Rightarrow \text{valid\_tuple } \text{tuple\_since}' \text{ tas}$ 
using valid_before by (auto simp add: ts_tuple_rel_def valid_tuple_def tuple_since'_def
 $\text{Mapping\_lookup\_upd\_set dest: Mapping\_keys\_dest split: option.splits})$ 
have valid_tuple_mono:  $\bigwedge \text{tas. valid\_tuple } \text{tuple\_since}' \text{ tas} \Rightarrow \text{valid\_tuple } \text{tuple\_since}' \text{ tas}$ 
by (auto simp add: valid_tuple_def tuple_since'_def Mapping_lookup_upd_set
 $\text{intro: Mapping\_keys\_intro split: option.splits})$ 
have ts_tuple_rel_auxlist':  $\text{ts\_tuple\_rel } (\text{set } \text{auxlist}') =$ 
 $\{\text{tas} \in \text{ts\_tuple\_rel } (\text{set } (\text{linearize } \text{data\_prev}) \cup \text{set } (\text{linearize } \text{data\_in}) \cup \{(\text{nt}, \text{X})\}).$ 
 $\text{valid\_tuple } \text{tuple\_since}' \text{ tas}\}$ 
proof (rule set_eqI, rule iffI)
fix tas
assume assm:  $\text{tas} \in \text{ts\_tuple\_rel } (\text{set } \text{auxlist}')$ 
show  $\text{tas} \in \{\text{tas} \in \text{ts\_tuple\_rel } (\text{set } (\text{linearize } \text{data\_prev}) \cup$ 
 $\text{set } (\text{linearize } \text{data\_in}) \cup \{(\text{nt}, \text{X})\}). \text{valid\_tuple } \text{tuple\_since}' \text{ tas}\}$ 
using assm[unfolded ts_tuple_rel_auxlist' ts_tuple_rel_Un]
proof (rule UnE)
assume assm:  $\text{tas} \in \text{ts\_tuple\_rel } (\text{set } \text{auxlist})$ 

```

```

then have  $tas \in \{tas \in ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in)).\ valid\_tuple\ tuple\_since\ tas\}$ 
using  $valid\_before$  by  $auto$ 
then show  $tas \in \{tas \in ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in) \cup \{(nt, X)\}).\ valid\_tuple\ tuple\_since'\ tas\}$ 
using  $assm$  by  $(auto\ simp\ only: ts\_tuple\_rel\_Un\ intro: valid\_tuple\_mono)$ 
next
assume  $assm: tas \in ts\_tuple\_rel \{(nt, X)\}$ 
show  $tas \in \{tas \in ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in) \cup \{(nt, X)\}).\ valid\_tuple\ tuple\_since'\ tas\}$ 
using  $assm\ valid\_before$  by  $(auto\ simp\ add: ts\_tuple\_rel\_Un\ tuple\_since'\_def$ 
   $valid\_tuple\_def\ Mapping\_lookup\_upd\_set\ ts\_tuple\_rel\_def\ dest: Mapping\_keys\_dest$ 
   $split: option.splits\ if\_splits)$ 
qed
next
fix  $tas$ 
assume  $assm: tas \in \{tas \in ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in) \cup \{(nt, X)\}).\ valid\_tuple\ tuple\_since'\ tas\}$ 
then have  $tas \in (ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in)) - ts\_tuple\_rel \{(nt, X)\}) \cup ts\_tuple\_rel \{(nt, X)\}$ 
by  $(auto\ simp\ only: ts\_tuple\_rel\_Un)$ 
then show  $tas \in ts\_tuple\_rel (set\ auxlist')$ 
proof  $(rule\ UnE)$ 
assume  $assm': tas \in ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in)) - ts\_tuple\_rel \{(nt, X)\}$ 
then have  $tas\_in: tas \in ts\_tuple\_rel (set (linearize\ data\_prev) \cup$ 
   $set (linearize\ data\_in))$ 
by  $(auto\ simp\ only: ts\_tuple\_rel\_def)$ 
obtain  $t\ as$  where  $tas\_def: tas = (t, as)$ 
by  $(cases\ tas)\ auto$ 
have  $t \in fst\ '(set (linearize\ data\_prev) \cup set (linearize\ data\_in))$ 
using  $assm'\ unfolding\ tas\_def$  by  $(force\ simp\ add: ts\_tuple\_rel\_def)$ 
then have  $t\_le\_nt: t \leq nt$ 
using  $valid\_before$  by  $auto$ 
have  $valid\_tas: valid\_tuple\ tuple\_since'\ tas$ 
using  $assm$  by  $auto$ 
have  $valid\_tuple\ tuple\_since\ tas$ 
proof  $(cases\ as \in Mapping.keys\ tuple\_since)$ 
case  $True$ 
then show  $?thesis$ 
using  $valid\_tas\ tas\_def$  by  $(auto\ simp\ add: valid\_tuple\_def\ tuple\_since'\_def$ 
   $Mapping\_lookup\_upd\_set\ split: option.splits\ if\_splits)$ 
next
case  $False$ 
then have  $t = nt\ as \in X$ 
using  $valid\_tas\ t\_le\_nt\ unfolding\ tas\_def$ 
by  $(auto\ simp\ add: valid\_tuple\_def\ tuple\_since'\_def\ Mapping\_lookup\_upd\_set$ 
   $intro: Mapping\_keys\_intro\ split: option.splits\ if\_splits)$ 
then have  $False$ 
using  $assm'\ unfolding\ tas\_def\ ts\_tuple\_rel\_def$  by  $(auto\ simp\ only: ts\_tuple\_rel\_def)$ 
then show  $?thesis$ 
by  $simp$ 
qed
then show  $tas \in ts\_tuple\_rel (set\ auxlist')$ 
using  $tas\_in\ valid\_before$  by  $(auto\ simp\ add: ts\_tuple\_rel\_auxlist')$ 
qed  $(auto\ simp\ only: ts\_tuple\_rel\_auxlist')$ 
qed
show  $?thesis$ 

```

```

proof (cases 0 ≥ left I)
  case True
  then have add_def: add_new_table_mmsaux args X (nt, gc, maskL, maskR, data_prev, data_in,
    tuple_in, tuple_since) = (nt, gc, maskL, maskR, data_prev, data_in',
    tuple_in', tuple_since')
  using data_in'_def tuple_in'_def tuple_since'_def unfolding I_def by auto
  have left_I: left I = 0
  using True by auto
  have ∀ t ∈ fst ' set (linearize data_in'). t ≤ nt ∧ nt - t ≥ left I
  using valid_before True by (auto simp add: lin_data_in')
  moreover have ∧ as. Mapping.lookup tuple_in' as = safe_max (fst '
    {tas ∈ ts_tuple_rel (set (linearize data_in')).
    valid_tuple tuple_since' tas ∧ as = snd tas})
  proof -
  fix as
  show Mapping.lookup tuple_in' as = safe_max (fst '
    {tas ∈ ts_tuple_rel (set (linearize data_in')).
    valid_tuple tuple_since' tas ∧ as = snd tas})
  proof (cases as ∈ X)
  case True
  have valid_tuple tuple_since' (nt, as)
  using True valid_before by (auto simp add: valid_tuple_def tuple_since'_def
    Mapping_lookup_upd_set dest: Mapping_keys_dest split: option.splits)
  moreover have (nt, as) ∈ ts_tuple_rel (insert (nt, X) (set (linearize data_in)))
  using True by (auto simp add: ts_tuple_rel_def)
  ultimately have nt_in: nt ∈ fst ' {tas ∈ ts_tuple_rel (insert (nt, X)
    (set (linearize data_in))). valid_tuple tuple_since' tas ∧ as = snd tas}
  proof -
  assume a1: valid_tuple tuple_since' (nt, as)
  assume (nt, as) ∈ ts_tuple_rel (insert (nt, X) (set (linearize data_in)))
  then have ∃ p. nt = fst p ∧ p ∈ ts_tuple_rel (insert (nt, X)
    (set (linearize data_in))) ∧ valid_tuple tuple_since' p ∧ as = snd p
  using a1 by simp
  then show nt ∈ fst ' {p ∈ ts_tuple_rel (insert (nt, X) (set (linearize data_in))).
    valid_tuple tuple_since' p ∧ as = snd p}
  by blast
  qed
  moreover have ∧ t. t ∈ fst ' {tas ∈ ts_tuple_rel (insert (nt, X)
    (set (linearize data_in))). valid_tuple tuple_since' tas ∧ as = snd tas} ⇒ t ≤ nt
  using valid_before by (auto split: option.splits)
  (metis (no_types, lifting) eq_imp_le fst_conv insertE ts_tuple_rel_dest)
  ultimately have Max (fst ' {tas ∈ ts_tuple_rel (set (linearize data_in)
    ∪ set [(nt, X)]). valid_tuple tuple_since' tas ∧ as = snd tas}) = nt
  using Max_eqI[OF finite_fst_ts_tuple_rel[of linearize data_in'],
    unfolded lin_data_in' set_append] by auto
  then show ?thesis
  using nt_in True
  by (auto simp add: tuple_in'_def Mapping_lookup_upd_set safe_max_def lin_data_in')
  next
  case False
  have {tas ∈ ts_tuple_rel (set (linearize data_in)).
    valid_tuple tuple_since' tas ∧ as = snd tas} =
  {tas ∈ ts_tuple_rel (set (linearize data_in')).
    valid_tuple tuple_since' tas ∧ as = snd tas}
  using False by (fastforce simp add: lin_data_in' ts_tuple_rel_def valid_tuple_def
    tuple_since'_def Mapping_lookup_upd_set intro: Mapping_keys_intro
    split: option.splits if_splits)
  then show ?thesis

```

```

    using valid_before False by (auto simp add: tuple_in'_def Mapping_lookup_upd_set)
  qed
qed
ultimately show ?thesis
  using assms table_auxlist' sorted_append[of map fst (linearize data_in)]
  lookup_tuple_since' ts_tuple_rel_auxlist' table_in' table_since'
  unfolding add_def auxlist'_def[symmetric] cur_nt I_def
  by (auto simp only: valid_mmsaux.simps lin_data_in' n_def R_def) auto
next
case False
then have add_def: add_new_table_mmsaux args X (nt, gc, maskL, maskR, data_prev, data_in,
  tuple_in, tuple_since) = (nt, gc, maskL, maskR, data_prev', data_in,
  tuple_in, tuple_since')
  using data_prev'_def tuple_since'_def unfolding I_def by auto
have left_I: left I > 0
  using False by auto
have  $\forall t \in \text{fst } \cdot \text{set } (\text{linearize data\_prev}'). t \leq nt \wedge nt - t < \text{left } I$ 
  using valid_before False by (auto simp add: lin_data_prev' I_def)
moreover have  $\bigwedge as. \{tas \in ts\_tuple\_rel (\text{set } (\text{linearize data\_in})).$ 
  valid_tuple tuple_since tas  $\wedge as = \text{snd } tas\} =$ 
 $\{tas \in ts\_tuple\_rel (\text{set } (\text{linearize data\_in})).$ 
  valid_tuple tuple_since' tas  $\wedge as = \text{snd } tas\}$ 
proof (rule set_eqI, rule iffI)
  fix as tas
  assume assm:  $tas \in \{tas \in ts\_tuple\_rel (\text{set } (\text{linearize data\_in})).$ 
    valid_tuple tuple_since' tas  $\wedge as = \text{snd } tas\}$ 
  then obtain t Z where Z_def:  $tas = (t, as) \wedge as \in Z \wedge (t, Z) \in \text{set } (\text{linearize data\_in})$ 
    valid_tuple tuple_since' (t, as)
    by (auto simp add: ts_tuple_rel_def)
  show  $tas \in \{tas \in ts\_tuple\_rel (\text{set } (\text{linearize data\_in})).$ 
    valid_tuple tuple_since tas  $\wedge as = \text{snd } tas\}$ 
  using assm
proof (cases  $as \in \text{Mapping.keys tuple\_since}$ )
  case False
  then have  $t \geq nt$ 
    using Z_def(4) by (auto simp add: valid_tuple_def tuple_since'_def
      Mapping_lookup_upd_set intro: Mapping_keys_intro split: option.splits if_splits)
  then show ?thesis
    using Z_def(3) valid_before left_I unfolding I_def by auto
qed (auto simp add: valid_tuple_def tuple_since'_def Mapping_lookup_upd_set
  dest: Mapping_keys_dest split: option.splits)
qed (auto simp add: Mapping_lookup_upd_set valid_tuple_def tuple_since'_def
  intro: Mapping_keys_intro split: option.splits)
ultimately show ?thesis
  using assms table_auxlist' sorted_append[of map fst (linearize data_prev)]
  False lookup_tuple_since' ts_tuple_rel_auxlist' table_in' table_since' wf_data_prev'
  valid_before
  unfolding add_def auxlist'_def[symmetric] cur_nt I_def
  by (auto simp only: valid_mmsaux.simps lin_data_prev' n_def R_def) fastforce+

qed
qed

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
using valid_add_new_table_mmsaux_unfolded assms
by (cases aux) fast

```

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 (*rule iffI*)

```

assume assm: as ∈ foldr (∪) (concat (map (λ(t, rel). if P t then [rel] else [])) auxlist) {}

```

```

then obtain t X where tX_def: P t as ∈ X (t, X) ∈ set auxlist

```

```

by (auto elim!: in_foldr_UnE)

```

```

then show ∃ t. (t, as) ∈ ts_tuple_rel (set auxlist) ∧ P t

```

```

by (auto simp add: ts_tuple_rel_def)

```

next

```

assume assm: ∃ t. (t, as) ∈ ts_tuple_rel (set auxlist) ∧ P t

```

```

then obtain t X where tX_def: P t as ∈ X (t, X) ∈ set auxlist

```

```

by (auto simp add: ts_tuple_rel_def)

```

```

show as ∈ foldr (∪) (concat (map (λ(t, rel). if P t then [rel] else [])) auxlist) {}

```

```

using in_foldr_UnI[OF tX_def(2)] tX_def assm by (induction auxlist) force+

```

qed

lemma *image_snd*: (a, b) ∈ X ⇒ b ∈ snd ` X

by *force*

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] {}

```

```

using valid_mmsaux_tuple_in_keys[OF assms] assms

```

```

by (auto simp add: image_Un ts_tuple_rel_Un foldr_ts_tuple_rel image_snd)

```

```

(fastforce intro: ts_tuple_rel_intro dest: ts_tuple_rel_dest)+

```

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] {}

```

```

using valid_result_mmsaux_unfolded by (cases aux) fast

```

interpretation *default_msaux*: msaux valid_mmsaux init_mmsaux add_new_ts_mmsaux gc_join_mmsaux

```

add_new_table_mmsaux result_mmsaux

```

```

using valid_init_mmsaux valid_add_new_ts_mmsaux valid_gc_join_mmsaux valid_add_new_table_mmsaux

```

```

valid_result_mmsaux

```

```

by unfold_locales assumption+

```

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 (λts'. ts' ≤ ts) (λtp'. tp' < tp) tstp

```

definition $tstp_le :: ts + tp \Rightarrow ts \Rightarrow tp \Rightarrow bool$ **where**
 $tstp_le\ tstp\ ts\ tp = case_sum\ (\lambda ts'.\ ts' \leq ts)\ (\lambda tp'.\ tp' \leq tp)\ tstp$

definition $ts_tp_lt :: ts \Rightarrow tp \Rightarrow ts + tp \Rightarrow bool$ **where**
 $ts_tp_lt\ ts\ tp\ tstp = case_sum\ (\lambda ts'.\ ts \leq ts')\ (\lambda tp'.\ tp < tp')\ tstp$

definition $ts_tp_lt' :: ts \Rightarrow tp \Rightarrow ts + tp \Rightarrow bool$ **where**
 $ts_tp_lt'\ ts\ tp\ tstp = case_sum\ (\lambda ts'.\ ts < ts')\ (\lambda tp'.\ tp \leq tp')\ tstp$

definition $ts_tp_le :: ts \Rightarrow tp \Rightarrow ts + tp \Rightarrow bool$ **where**
 $ts_tp_le\ ts\ tp\ tstp = case_sum\ (\lambda ts'.\ ts \leq ts')\ (\lambda tp'.\ tp \leq tp')\ tstp$

fun $max_tstp :: ts + tp \Rightarrow ts + tp \Rightarrow ts + tp$ **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 $max_tstp_idem: max_tstp\ (max_tstp\ x\ y)\ y = max_tstp\ x\ y$
by $(cases\ x; cases\ y)\ auto$

lemma $max_tstp_idem': max_tstp\ x\ (max_tstp\ x\ y) = max_tstp\ x\ y$
by $(cases\ x; cases\ y)\ auto$

lemma $max_tstp_d\ d: max_tstp\ d\ d = d$
by $(cases\ d)\ auto$

lemma $max_cases: (max\ a\ b = a \Longrightarrow P) \Longrightarrow (max\ a\ b = b \Longrightarrow P) \Longrightarrow P$
by $(metis\ max_def)$

lemma $max_tstpE: isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow (max_tstp\ tstp\ tstp' = tstp \Longrightarrow P) \Longrightarrow$
 $(max_tstp\ tstp\ tstp' = tstp' \Longrightarrow P) \Longrightarrow P$
by $(cases\ tstp; cases\ tstp')\ (auto\ elim: max_cases)$

lemma $max_tstp_intro: tstp_lt\ tstp\ ts\ tp \Longrightarrow tstp_lt\ tstp'\ ts\ tp \Longrightarrow isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow$
 $tstp_lt\ (max_tstp\ tstp\ tstp')\ ts\ tp$
by $(auto\ simp\ add: tstp_lt_def\ split: sum.splits)$

lemma $max_tstp_intro': isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow$
 $ts_tp_le\ ts'\ tp'\ tstp \Longrightarrow ts_tp_le\ ts'\ tp'\ (max_tstp\ tstp\ tstp')$
by $(cases\ tstp; cases\ tstp')\ (auto\ simp\ add: ts_tp_le_def\ tstp_le_def\ split: sum.splits)$

lemma $max_tstp_intro'': isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow$
 $ts_tp_le\ ts'\ tp'\ tstp' \Longrightarrow ts_tp_le\ ts'\ tp'\ (max_tstp\ tstp\ tstp')$
by $(cases\ tstp; cases\ tstp')\ (auto\ simp\ add: ts_tp_le_def\ tstp_le_def\ split: sum.splits)$

lemma $max_tstp_intro''': isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow$
 $ts_tp_lt'\ ts'\ tp'\ tstp \Longrightarrow ts_tp_lt'\ ts'\ tp'\ (max_tstp\ tstp\ tstp')$
by $(cases\ tstp; cases\ tstp')\ (auto\ simp\ add: ts_tp_lt'_def\ tstp_le_def\ split: sum.splits)$

lemma $max_tstp_intro''': isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow$
 $ts_tp_lt'\ ts'\ tp'\ tstp' \Longrightarrow ts_tp_lt'\ ts'\ tp'\ (max_tstp\ tstp\ tstp')$
by $(cases\ tstp; cases\ tstp')\ (auto\ simp\ add: ts_tp_lt'_def\ tstp_le_def\ split: sum.splits)$

lemma $max_tstp_isl: isl\ tstp \longleftrightarrow isl\ tstp' \Longrightarrow isl\ (max_tstp\ tstp\ tstp') \longleftrightarrow isl\ tstp$
by $(cases\ tstp; cases\ tstp')\ auto$

definition $filter_a1_map :: bool \Rightarrow tp \Rightarrow ('a\ tuple, tp)\ mapping \Rightarrow 'a\ table$ **where**

filter_a1_map pos *tp a1_map* =
 $\{xs \in \text{Mapping.keys } a1_map. \text{ case Mapping.lookup } a1_map \text{ } xs \text{ of Some } tp' \Rightarrow$
 $(pos \longrightarrow tp' \leq tp) \wedge (\neg pos \longrightarrow tp \leq tp')\}$

definition *filter_a2_map* :: $\mathcal{I} \Rightarrow ts \Rightarrow tp \Rightarrow (tp, ('a \text{ tuple}, ts + tp) \text{ mapping}) \text{ mapping} \Rightarrow$
 $'a \text{ table}$ **where**
filter_a2_map *I ts tp a2_map* = $\{xs. \exists tp' \leq tp. (\text{case Mapping.lookup } a2_map \text{ } tp' \text{ of Some } m \Rightarrow$
 $(\text{case Mapping.lookup } m \text{ } xs \text{ of Some } tstp \Rightarrow ts_tp_lt' \text{ } ts \text{ } tp \text{ } tstp \mid _ \Rightarrow \text{False})$
 $\mid _ \Rightarrow \text{False})\}$

fun *triple_eq_pair* :: $('a \times 'b \times 'c) \Rightarrow ('a \times 'd) \Rightarrow ('d \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'd \Rightarrow 'c) \Rightarrow \text{bool}$ **where**
triple_eq_pair (*t, a1, a2*) (*ts', tp'*) *f g* $\longleftrightarrow t = ts' \wedge a1 = f \text{ } tp' \wedge a2 = g \text{ } ts' \text{ } tp'$

fun *valid_mmuaux'* :: $\text{args} \Rightarrow ts \Rightarrow ts \Rightarrow 'a \text{ mmuaux} \Rightarrow 'a \text{ muaux} \Rightarrow \text{bool}$ **where**
valid_mmuaux' *args cur dt (tp, tss, len, maskL, maskR, a1_map, a2_map,*
done, done_length) auxlist \longleftrightarrow
 $\text{args_L } \text{args} \subseteq \text{args_R } \text{args} \wedge$
 $\text{maskL} = \text{join_mask } (\text{args_n } \text{args}) (\text{args_L } \text{args}) \wedge$
 $\text{maskR} = \text{join_mask } (\text{args_n } \text{args}) (\text{args_R } \text{args}) \wedge$
 $\text{len} \leq tp \wedge$
 $\text{length } (\text{linearize } tss) = \text{len} \wedge \text{sorted } (\text{linearize } tss) \wedge$
 $(\forall t \in \text{set } (\text{linearize } tss). t \leq \text{cur} \wedge \text{enat } t + \text{right } (\text{args_ivl } \text{args})) \wedge$
 $\text{table } (\text{args_n } \text{args}) (\text{args_L } \text{args}) (\text{Mapping.keys } a1_map) \wedge$
 $\text{Mapping.keys } a2_map = \{tp - \text{len}..tp\} \wedge$
 $(\forall xs \in \text{Mapping.keys } a1_map. \text{ case Mapping.lookup } a1_map \text{ } xs \text{ of Some } tp' \Rightarrow tp' < tp) \wedge$
 $(\forall tp' \in \text{Mapping.keys } a2_map. \text{ case Mapping.lookup } a2_map \text{ } tp' \text{ of Some } m \Rightarrow$
 $\text{table } (\text{args_n } \text{args}) (\text{args_R } \text{args}) (\text{Mapping.keys } m) \wedge$
 $(\forall xs \in \text{Mapping.keys } m. \text{ case Mapping.lookup } m \text{ } xs \text{ of Some } tstp \Rightarrow$
 $tstp_lt \text{ } tstp (\text{cur} - (\text{left } (\text{args_ivl } \text{args}) - 1)) \text{ } tp \wedge (\text{isl } tstp \longleftrightarrow \text{left } (\text{args_ivl } \text{args}) > 0))) \wedge$
 $\text{length } \text{done} = \text{done_length} \wedge \text{length } \text{done} + \text{len} = \text{length } \text{auxlist} \wedge$
 $\text{rev } \text{done} = \text{map } \text{proj_thd } (\text{take } (\text{length } \text{done}) \text{ } \text{auxlist}) \wedge$
 $(\forall x \in \text{set } (\text{take } (\text{length } \text{done}) \text{ } \text{auxlist}). \text{check_before } (\text{args_ivl } \text{args}) \text{ } dt \text{ } x) \wedge$
 $\text{sorted } (\text{map } \text{fst } \text{auxlist}) \wedge$
 $\text{list_all2 } (\lambda x y. \text{triple_eq_pair } x \text{ } y (\lambda tp'. \text{filter_a1_map } (\text{args_pos } \text{args}) \text{ } tp' \text{ } a1_map)$
 $(\lambda ts' tp'. \text{filter_a2_map } (\text{args_ivl } \text{args}) \text{ } ts' \text{ } tp' \text{ } a2_map)) (\text{drop } (\text{length } \text{done}) \text{ } \text{auxlist})$
 $(\text{zip } (\text{linearize } tss) [tp - \text{len}..<tp])$

definition *valid_mmuaux* :: $\text{args} \Rightarrow ts \Rightarrow 'a \text{ mmuaux} \Rightarrow 'a \text{ muaux} \Rightarrow$
 bool **where**
valid_mmuaux *args cur* = *valid_mmuaux'* *args cur cur*

fun *eval_step_mmuaux* :: $'a \text{ mmuaux} \Rightarrow 'a \text{ mmuaux}$ **where**
eval_step_mmuaux (*tp, tss, len, maskL, maskR, a1_map, a2_map,*
done, done_length) = $(\text{case safe_hd } tss \text{ of (Some } ts, tss') \Rightarrow$
 $(\text{case Mapping.lookup } a2_map \text{ } (tp - \text{len}) \text{ of Some } m \Rightarrow$
 $\text{let } m = \text{Mapping.filter } (\lambda _ \text{ } tstp. \text{ts_tp_lt}' \text{ } ts \text{ } (tp - \text{len}) \text{ } tstp) \text{ } m;$
 $T = \text{Mapping.keys } m;$
 $a2_map = \text{Mapping.update } (tp - \text{len} + 1)$
 $(\text{case Mapping.lookup } a2_map \text{ } (tp - \text{len} + 1) \text{ of Some } m' \Rightarrow$
 $\text{Mapping.combine } (\lambda \text{ } tstp \text{ } tstp'. \text{max_tstp } \text{ } tstp \text{ } tstp') \text{ } m \text{ } m') \text{ } a2_map;$
 $a2_map = \text{Mapping.delete } (tp - \text{len}) \text{ } a2_map \text{ in}$
 $(tp, \text{tl_queue } tss', \text{len} - 1, \text{maskL}, \text{maskR}, a1_map, a2_map,$
 $T \# \text{done}, \text{done_length} + 1)))$

lemma *Mapping_update_keys*: $\text{Mapping.keys } (\text{Mapping.update } a \text{ } b \text{ } m) = \text{Mapping.keys } m \cup \{a\}$
by *transfer auto*

lemma *drop_is_Cons_take*: $\text{drop } n \text{ } xs = y \# ys \implies \text{take } (\text{Suc } n) \text{ } xs = \text{take } n \text{ } xs @ [y]$

proof (induction xs arbitrary: n)

case Nil

then show ?case by simp

next

case (Cons x xs)

then show ?case by (cases n) simp_all

qed

lemma list_all2_weaken: list_all2 f xs ys \implies

$(\bigwedge x y. (x, y) \in \text{set } (\text{zip } xs \text{ } ys) \implies f x y \implies f' x y) \implies \text{list_all2 } f' \text{ } xs \text{ } ys$

by (induction xs ys rule: list_all2_induct) auto

lemma Mapping_lookup_delete: Mapping.lookup (Mapping.delete k m) k' =

(if k = k' then None else Mapping.lookup m k')

by transfer auto

lemma Mapping_lookup_update: Mapping.lookup (Mapping.update k v m) k' =

(if k = k' then Some v else Mapping.lookup m k')

by transfer auto

lemma hd_le_set: sorted xs $\implies x \in \text{set } xs \implies \text{hd } xs \leq x$

by (metis dual_order.eq_iff equals0D hd_Cons_tl set_ConsD set_empty sorted_simps(2))

lemma Mapping_lookup_combineE: Mapping.lookup (Mapping.combine f m m') k = Some v \implies

(Mapping.lookup m k = Some v \implies P) \implies

(Mapping.lookup m' k = Some v \implies P) \implies

$(\bigwedge v' v''. \text{Mapping.lookup } m \text{ } k = \text{Some } v' \implies \text{Mapping.lookup } m' \text{ } k = \text{Some } v'' \implies$

$f v' v'' = v \implies P) \implies P$

unfolding Mapping.lookup_combine

by (auto simp add: combine_options_def split: option.splits)

lemma Mapping_keys_filterI: Mapping.lookup m k = Some v $\implies f k v \implies$

$k \in \text{Mapping.keys } (\text{Mapping.filter } f \text{ } m)$

by transfer (auto split: option.splits if_splits)

lemma Mapping_keys_filterD: $k \in \text{Mapping.keys } (\text{Mapping.filter } f \text{ } m) \implies$

$\exists v. \text{Mapping.lookup } m \text{ } k = \text{Some } v \wedge f k v$

by transfer (auto split: option.splits if_splits)

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

lin_ts_mmuaux aux = ts # tss'' enat ts + right (args_ivl args) < dt

shows valid_mmuaux' args cur dt (eval_step_mmuaux aux) auxlist \wedge

lin_ts_mmuaux (eval_step_mmuaux aux) = tss''

proof -

define I **where** I = args_ivl args

define n **where** n = args_n args

define L **where** L = args_L args

define R **where** R = args_R args

define pos **where** pos = args_pos args

obtain tp len tss maskL maskR a1_map a2_map done done_length **where** aux_def:

aux = (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length)

by (cases aux) auto

then obtain tss' **where** safe_hd_eq: safe_hd tss = (Some ts, tss')

```

using assms(2) safe_hd_rep case_optionE
by (cases safe_hd tss) fastforce
note valid_before = assms(1)[unfolded aux_def]
have lin_tss_not_Nil: linearize tss ≠ []
using safe_hd_rep[OF safe_hd_eq] by auto
have ts_hd: ts = hd (linearize tss)
using safe_hd_rep[OF safe_hd_eq] by auto
have lin_tss': linearize tss' = linearize tss
using safe_hd_rep[OF safe_hd_eq] by auto
have tss'_not_empty: ¬is_empty tss'
using is_empty_alt[of tss'] lin_tss_not_Nil unfolding lin_tss' by auto
have len_pos: len > 0
using lin_tss_not_Nil valid_before by auto
have a2_map_keys: Mapping.keys a2_map = {tp - len..tp}
using valid_before by auto
have len_tp: len ≤ tp
using valid_before by auto
have tp_minus_keys: tp - len ∈ Mapping.keys a2_map
using a2_map_keys by auto
have tp_minus_keys': tp - len + 1 ∈ Mapping.keys a2_map
using a2_map_keys len_pos len_tp by auto
obtain m where m_def: Mapping.lookup a2_map (tp - len) = Some m
using tp_minus_keys by (auto dest: Mapping_keys_dest)
have table n R (Mapping.keys m)
  (∀xs ∈ Mapping.keys m. case Mapping.lookup m xs of Some tstp ⇒
tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ⇔ left I > 0))
using tp_minus_keys m_def valid_before
unfolding valid_mmuaux'.simps n_def I_def R_def by fastforce+
then have m_inst: table n R (Mapping.keys m)
  ∧xs tstp. Mapping.lookup m xs = Some tstp ⇒
tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ⇔ left I > 0)
using Mapping_keys_intro by fastforce+
have m_inst_isl: ∧xs tstp. Mapping.lookup m xs = Some tstp ⇒ isl tstp ⇔ left I > 0
using m_inst(2) by fastforce
obtain m' where m'_def: Mapping.lookup a2_map (tp - len + 1) = Some m'
using tp_minus_keys' by (auto dest: Mapping_keys_dest)
have table n R (Mapping.keys m')
  (∀xs ∈ Mapping.keys m'. case Mapping.lookup m' xs of Some tstp ⇒
tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ⇔ left I > 0))
using tp_minus_keys' m'_def valid_before
unfolding valid_mmuaux'.simps I_def n_def R_def by fastforce+
then have m'_inst: table n R (Mapping.keys m')
  ∧xs tstp. Mapping.lookup m' xs = Some tstp ⇒
tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ⇔ left I > 0)
using Mapping_keys_intro by fastforce+
have m'_inst_isl: ∧xs tstp. Mapping.lookup m' xs = Some tstp ⇒ isl tstp ⇔ left I > 0
using m'_inst(2) by fastforce
define m_upd where m_upd = Mapping.filter (λ_ tstp. ts_tp_lt' ts (tp - len) tstp) m
define T where T = Mapping.keys m_upd
define mc where mc = Mapping.combine (λtstp tstp'. max_tstp tstp tstp') m_upd m'
define a2_map' where a2_map' = Mapping.update (tp - len + 1) mc a2_map
define a2_map'' where a2_map'' = Mapping.delete (tp - len) a2_map'
have m_upd_lookup: ∧xs tstp. Mapping.lookup m_upd xs = Some tstp ⇒
tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ⇔ left I > 0)
unfolding m_upd_def Mapping.lookup_filter
using m_inst(2) by (auto split: option.splits if_splits)
have mc_lookup: ∧xs tstp. Mapping.lookup mc xs = Some tstp ⇒
tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ⇔ left I > 0)

```

```

unfolding mc_def Mapping.lookup_combine
using m_upd_lookup m'_inst(2)
by (auto simp add: combine_options_def max_tstp_isl intro: max_tstp_intro split: option.splits)
have mc_keys: Mapping.keys mc  $\subseteq$  Mapping.keys m  $\cup$  Mapping.keys m'
unfolding mc_def Mapping.keys_combine m_upd_def
using Mapping.keys_filter by fastforce
have tp_len_assoc:  $tp - len + 1 = tp - (len - 1)$ 
using len_pos len_tp by auto
have a2_map''_keys: Mapping.keys a2_map'' =  $\{tp - (len - 1)..tp\}$ 
unfolding a2_map''_def a2_map'_def Mapping.keys_delete Mapping_update_keys a2_map_keys
using tp_len_assoc by auto
have lin_tss_Cons: linearize tss = ts # linearize (tl_queue tss')
using lin_tss_not_Nil
by (auto simp add: tl_queue_rep[OF tss'_not_empty] lin_tss' ts_hd)
have tp_len_tp_unfold:  $[tp - len..<tp] = (tp - len) \# [tp - (len - 1)..<tp]$ 
unfolding tp_len_assoc[symmetric]
using len_pos len_tp Suc_diff_le upt_conv_Cons by auto
have id:  $\bigwedge x. x \in \{tp - (len - 1) + 1..tp\} \implies$ 
  Mapping.lookup a2_map'' x = Mapping.lookup a2_map x
unfolding a2_map''_def a2_map'_def Mapping_lookup_delete Mapping_lookup_update tp_len_assoc
using len_tp by auto
have list_all2: list_all2 ( $\lambda x y. triple\_eq\_pair\ x\ y\ (\lambda tp'. filter\_a1\_map\ pos\ tp'\ a1\_map)$ )
  ( $\lambda ts' tp'. filter\_a2\_map\ I\ ts'\ tp'\ a2\_map$ )
  (drop (length done) auxlist) (zip (linearize tss) [tp - len..<tp])
using valid_before unfolding I_def pos_def by auto
obtain hd_aux tl_aux where aux_split: drop (length done) auxlist = hd_aux # tl_aux
  case hd_aux of (t, a1, a2)  $\Rightarrow$  (t, a1, a2) =
  (ts, filter_a1_map pos (tp - len) a1_map, filter_a2_map I ts (tp - len) a2_map)
and list_all2': list_all2 ( $\lambda x y. triple\_eq\_pair\ x\ y\ (\lambda tp'. filter\_a1\_map\ pos\ tp'\ a1\_map)$ )
  ( $\lambda ts' tp'. filter\_a2\_map\ I\ ts'\ tp'\ a2\_map$ ) tl_aux
  (zip (linearize (tl_queue tss')) [tp - (len - 1)..<tp])
using list_all2[unfolded lin_tss_Cons tp_len_tp_unfold zip_Cons_Cons list_all2_Cons2] by auto
have lookup''_tp_minus: Mapping.lookup a2_map'' (tp - (len - 1)) = Some mc
unfolding a2_map''_def a2_map'_def Mapping_lookup_delete Mapping_lookup_update
  tp_len_assoc[symmetric]
using len_tp by auto
have filter_a2_map_cong:  $\bigwedge ts' tp'. ts' \in set\ (linearize\ tss) \implies$ 
   $tp' \in \{tp - (len - 1)..<tp\} \implies filter\_a2\_map\ I\ ts'\ tp'\ a2\_map =$ 
   $filter\_a2\_map\ I\ ts'\ tp'\ a2\_map''$ 
proof (rule set_eqI, rule iffI)
  fix ts' tp' xs
  assume assms:  $ts' \in set\ (linearize\ tss)$ 
   $tp' \in \{tp - (len - 1)..<tp\}$   $xs \in filter\_a2\_map\ I\ ts'\ tp'\ a2\_map$ 
obtain tp_bef m_bef tstp where defs:  $tp\_bef \leq tp'$   $Mapping.lookup\ a2\_map\ tp\_bef = Some\ m\_bef$ 
   $Mapping.lookup\ m\_bef\ xs = Some\ tstp\ ts\_tp\_lt'\ ts'\ tp'\ tstp$ 
using assms(3)[unfolded filter_a2_map_def]
by (fastforce split: option.splits)
have ts_le_ts':  $ts \leq ts'$ 
using hd_le_set[OF _ assms(1)] valid_before
unfolding ts_hd by auto
have tp_bef_in:  $tp\_bef \in \{tp - len..tp\}$ 
using defs(2) valid_before by (auto intro!: Mapping_keys_intro)
have tp_minus_le:  $tp - (len - 1) \leq tp'$ 
using assms(2) by auto
show  $xs \in filter\_a2\_map\ I\ ts'\ tp'\ a2\_map''$ 
proof (cases  $tp\_bef \leq tp - (len - 1)$ )
  case True
  show ?thesis

```

```

proof (cases tp_bef = tp - len)
  case True
  have m_bef_m: m_bef = m
    using defs(2) m_def
    unfolding True by auto
  have Mapping.lookup m_upd xs = Some tstp
    using defs(3,4) assms(2) ts_le_ts' unfolding m_bef_m m_upd_def
    by (auto simp add: Mapping.lookup_filter ts_tp_lt'_def intro: Mapping_keys_intro
      split: sum.splits)
  then have case Mapping.lookup mc xs of None  $\Rightarrow$  False | Some tstp  $\Rightarrow$ 
    ts_tp_lt' ts' tp' tstp
    unfolding mc_def Mapping.lookup_combine
    using m'_inst(2) m_upd_lookup
    by (auto simp add: combine_options_def defs(4) intro!: max_tstp_intro'''
      dest: Mapping_keys_dest split: option.splits)
  then show ?thesis
    using lookup''_tp_minus tp_minus_le defs
    unfolding m_bef_m filter_a2_map_def by (auto split: option.splits)
next
  case False
  then have tp_bef = tp - (len - 1)
    using True tp_bef_in by auto
  then have m_bef_m: m_bef = m'
    using defs(2) m'_def
    unfolding tp_len_assoc by auto
  have case Mapping.lookup mc xs of None  $\Rightarrow$  False | Some tstp  $\Rightarrow$ 
    ts_tp_lt' ts' tp' tstp
    unfolding mc_def Mapping.lookup_combine
    using m'_inst(2) m_upd_lookup defs(3)[unfolded m_bef_m]
    by (auto simp add: combine_options_def defs(4) intro!: max_tstp_intro''''
      dest: Mapping_keys_dest split: option.splits)
  then show ?thesis
    using lookup''_tp_minus tp_minus_le defs
    unfolding m_bef_m filter_a2_map_def by (auto split: option.splits)
qed
next
  case False
  then have Mapping.lookup a2_map'' tp_bef = Mapping.lookup a2_map tp_bef
    using id tp_bef_in len_tp by auto
  then show ?thesis
    unfolding filter_a2_map_def
    using defs by auto
qed
next
  fix ts' tp' xs
  assume assms: ts'  $\in$  set (linearize tss) tp'  $\in$  {tp - (len - 1)..<tp}
    xs  $\in$  filter_a2_map I ts' tp' a2_map''
  obtain tp_bef m_bef tstp where defs: tp_bef  $\leq$  tp'
    Mapping.lookup a2_map'' tp_bef = Some m_bef
    Mapping.lookup m_bef xs = Some tstp ts_tp_lt' ts' tp' tstp
  using assms(3)[unfolded filter_a2_map_def]
  by (fastforce split: option.splits)
  have ts_le_ts': ts  $\leq$  ts'
    using hd_le_set[OF _ assms(1)] valid_before
    unfolding ts_hd by auto
  have tp_bef_in: tp_bef  $\in$  {tp - (len - 1)..tp}
    using defs(2) a2_map''_keys by (auto intro!: Mapping_keys_intro)
  have tp_minus_le: tp - len  $\leq$  tp' tp - (len - 1)  $\leq$  tp'

```

```

using assms(2) by auto
show  $xs \in \text{filter\_a2\_map } I \text{ ts' tp' a2\_map}$ 
proof (cases  $tp\_bef = tp - (len - 1)$ )
  case True
    have  $m\_beg\_mc: m\_bef = mc$ 
      using defs(2)
      unfolding True a2\_map''\_def a2\_map'\_def tp\_len\_assoc Mapping\_lookup\_delete
        Mapping.lookup\_update
      by (auto split: if\_splits)
    show ?thesis
      using defs(3)[unfolded m\_beg\_mc mc\_def]
    proof (rule Mapping\_lookup\_combineE)
      assume lassm: Mapping.lookup m\_upd xs = Some tstp
      then show  $xs \in \text{filter\_a2\_map } I \text{ ts' tp' a2\_map}$ 
        unfolding m\_upd\_def Mapping.lookup\_filter
        using m\_def tp\_minus\_le(1) defs
        by (auto simp add: filter\_a2\_map\_def split: option.splits if\_splits)
      next
        assume lassm: Mapping.lookup m' xs = Some tstp
        then show  $xs \in \text{filter\_a2\_map } I \text{ ts' tp' a2\_map}$ 
          using m'\_def defs(4) tp\_minus\_le defs
          unfolding filter\_a2\_map\_def tp\_len\_assoc
          by auto
        next
          fix v' v''
          assume lassms: Mapping.lookup m\_upd xs = Some v' Mapping.lookup m' xs = Some v''
            max\_tstp v' v'' = tstp
          show  $xs \in \text{filter\_a2\_map } I \text{ ts' tp' a2\_map}$ 
          proof (rule max\_tstpE)
            show isl v' = isl v''
              using lassms(1,2) m\_upd\_lookup m'\_inst(2)
              by auto
            next
              assume max\_tstp v' v'' = v'
              then show  $xs \in \text{filter\_a2\_map } I \text{ ts' tp' a2\_map}$ 
                using lassms(1,3) m\_def defs tp\_minus\_le(1)
                unfolding tp\_len\_assoc m\_upd\_def Mapping.lookup\_filter
                by (auto simp add: filter\_a2\_map\_def split: option.splits if\_splits)
              next
                assume max\_tstp v' v'' = v''
                then show  $xs \in \text{filter\_a2\_map } I \text{ ts' tp' a2\_map}$ 
                  using lassms(2,3) m'\_def defs tp\_minus\_le(2)
                  unfolding tp\_len\_assoc
                  by (auto simp add: filter\_a2\_map\_def)
                qed
              qed
            next
              case False
              then have Mapping.lookup a2\_map'' tp\_bef = Mapping.lookup a2\_map tp\_bef
                using id tp\_bef\_in by auto
              then show ?thesis
                unfolding filter\_a2\_map\_def
                using defs by auto (metis option.simps(5))
              qed
            qed
          have set tl\_tss': set (linearize (tl\_queue tss'))  $\subseteq$  set (linearize tss)
            unfolding tl\_queue\_rep[OF tss'\_not\_empty] lin\_tss Cons by auto
          have list\_all2'': list\_all2 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map } \text{pos } tp' \text{ a1\_map})$ )

```

```

    (λts' tp'. filter_a2_map I ts' tp' a2_map'') tl_aux
    (zip (linearize (tl_queue tss')) [tp - (len - 1)..<tp])
using filter_a2_map_cong set_tl_tss'
by (intro list_all2_weaken[OF list_all2']) (auto elim!: in_set_zipE split: prod.splits)
have lookup'': ∀ tp' ∈ Mapping.keys a2_map''. case Mapping.lookup a2_map'' tp' of Some m ⇒
    table n R (Mapping.keys m) ∧ (∀ xs ∈ Mapping.keys m. case Mapping.lookup m xs of Some tstp ⇒
    tstp_lt tstp (cur - (left I - 1)) tp ∧ isl tstp = (0 < left I))
proof (rule ballI)
  fix tp'
  assume assm: tp' ∈ Mapping.keys a2_map''
  then obtain f where f_def: Mapping.lookup a2_map'' tp' = Some f
    by (auto dest: Mapping_keys_dest)
  have tp'_in: tp' ∈ {tp - (len - 1)..tp}
    using assm unfolding a2_map''_keys .
  then have tp'_in_keys: tp' ∈ Mapping.keys a2_map
    using valid_before by auto
  have table n R (Mapping.keys f) ∧
    (∀ xs ∈ Mapping.keys f. case Mapping.lookup f xs of Some tstp ⇒
    tstp_lt tstp (cur - (left I - 1)) tp ∧ isl tstp = (0 < left I))
  proof (cases tp' = tp - (len - 1))
    case True
      then have f_mc: f = mc
        using f_def
      unfolding a2_map''_def a2_map'_def Mapping_lookup_delete Mapping_lookup_update tp_len_assoc
        by (auto split: if_splits)
      have table n R (Mapping.keys f)
        unfolding f_mc
        using mc_keys m_def m'_def m_inst m'_inst
        by (auto simp add: table_def)
      moreover have ∀ xs ∈ Mapping.keys f. case Mapping.lookup f xs of Some tstp ⇒
        tstp_lt tstp (cur - (left I - 1)) tp ∧ isl tstp = (0 < left I)
        using assm Mapping_keys_filter m_inst(2) m_inst_isl m'_inst(2) m'_inst_isl max_tstp_isl
        unfolding f_mc mc_def Mapping_lookup_combine
        by (auto simp add: combine_options_def m_upd_def Mapping_lookup_filter
          intro!: max_tstp_intro Mapping_keys_intro dest!: Mapping_keys_dest
          split: option.splits)
      ultimately show ?thesis
        by auto
    case False
      have Mapping.lookup a2_map tp' = Some f
        using tp'_in id[of tp'] f_def by auto
      then show ?thesis
        using tp'_in_keys valid_before
        unfolding valid_mmuaux'.simps I_def n_def R_def by fastforce
  qed
then show case Mapping.lookup a2_map'' tp' of Some m ⇒
    table n R (Mapping.keys m) ∧ (∀ xs ∈ Mapping.keys m. case Mapping.lookup m xs of Some tstp ⇒
    tstp_lt tstp (cur - (left I - 1)) tp ∧ isl tstp = (0 < left I))
    using f_def by auto
  qed
have tl_aux_def: tl_aux = drop (length done + 1) auxlist
  using aux_split(1) by (metis Suc_eq_plus1 add_Suc drop0 drop_Suc_Cons drop_drop)
have T_eq: T = filter_a2_map I ts (tp - len) a2_map
proof (rule set_eqI, rule iffI)
  fix xs
  assume xs ∈ filter_a2_map I ts (tp - len) a2_map
  then obtain tp_bef m_bef tstp where defs: tp_bef ≤ tp - len

```

```

    Mapping.lookup a2_map tp_bef = Some m_bef
    Mapping.lookup m_bef xs = Some tstp ts_tp_lt' ts (tp - len) tstp
  by (fastforce simp add: filter_a2_map_def split: option.splits)
  then have tp_bef_minus: tp_bef = tp - len
    using valid_before Mapping_keys_intro by force
  have m_bef_m: m_bef = m
    using defs(2) m_def
    unfolding tp_bef_minus by auto
  show xs ∈ T
    using defs
    unfolding T_def m_upd_def m_bef_m
    by (auto intro: Mapping_keys_filterI Mapping_keys_intro)
next
fix xs
assume xs ∈ T
then show xs ∈ filter_a2_map I ts (tp - len) a2_map
  using m_def Mapping_keys_filter
  unfolding T_def m_upd_def filter_a2_map_def
  by (auto simp add: filter_a2_map_def dest!: Mapping_keys_filterD split: if_splits)
qed
have min_auxlist_done: min (length auxlist) (length done) = length done
  using valid_before by auto
then have ∀ x ∈ set (take (length done) auxlist). check_before I dt x
  rev done = map proj_thd (take (length done) auxlist)
  using valid_before unfolding I_def by auto
then have list_all': (∀ x ∈ set (take (length (T # done)) auxlist). check_before I dt x)
  rev (T # done) = map proj_thd (take (length (T # done)) auxlist)
  using drop_is_Cons_take[OF aux_split(1)] aux_split(2) assms(3)
  by (auto simp add: T_eq I_def)
have eval_step_mmuaux_eq: eval_step_mmuaux (tp, tss, len, maskL, maskR, a1_map, a2_map,
  done, done_length) = (tp, tl_queue tss', len - 1, maskL, maskR, a1_map, a2_map'',
  T # done, done_length + 1)
  using safe_hd_eq m'_def m_upd_def T_def mc_def a2_map'_def a2_map''_def
  by (auto simp add: Let_def)
have lin_ts_mmuaux (eval_step_mmuaux aux) = tss''
  using lin_tss_Cons assms(2) unfolding aux_def eval_step_mmuaux_eq by auto
then show ?thesis
  using valid_before a2_map''_keys sorted_tl list_all' lookup'' list_all2''
  unfolding eval_step_mmuaux_eq valid_mmuaux'.simps tl_aux_def aux_def I_def n_def R_def
  pos_def
  using lin_tss_not_Nil safe_hd_eq len_pos
  by (auto simp add: list.set_sel(2) lin_tss' tl_queue_rep[OF tss'_not_empty] min_auxlist_done)
qed

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)
  using assms sorted_wrt_drop by (auto simp add: drop_map[symmetric])

lemma valid_mmuaux'_mono:
  assumes valid_mmuaux' args cur dt aux auxlist dt ≤ dt'
  shows valid_mmuaux' args cur dt' aux auxlist
  using assms less_le_trans by (cases aux) fastforce

lemma valid_foldl_eval_step_mmuaux':

```

```

assumes valid_before: valid_mmuaux' args cur dt aux auxlist
  lin_ts_mmuaux aux = tss @ tss'
   $\bigwedge ts. ts \in \text{set } (\text{take } (\text{length } tss) (\text{lin\_ts\_mmuaux } aux)) \implies \text{enat } ts + \text{right } (\text{args\_ivl } args) < dt$ 
shows valid_mmuaux' args cur dt (foldl (\lambda aux _. eval_step_mmuaux aux) aux tss) auxlist  $\wedge$ 
  lin_ts_mmuaux (foldl (\lambda aux _. eval_step_mmuaux aux) aux tss) = tss'
using assms
proof (induction tss arbitrary: aux)
case (Cons ts tss)
  have app_ass: lin_ts_mmuaux aux = ts # (tss @ tss')
  using Cons(3) by auto
  have enat ts + right (args_ivl args) < dt
  using Cons by auto
  then have valid_step: valid_mmuaux' args cur dt (eval_step_mmuaux aux) auxlist
  lin_ts_mmuaux (eval_step_mmuaux aux) = tss @ tss'
  using valid_eval_step_mmuaux'[OF Cons(2) app_ass] by auto
  show ?case
  using Cons(1)[OF valid_step] valid_step Cons(4) app_ass by auto
qed auto

lemma sorted_dropWhile_filter: sorted xs \implies dropWhile (\lambda t. enat t + right I < enat nt) xs =
  filter (\lambda t. \neg enat t + right I < enat nt) xs
proof (induction xs)
case (Cons x xs)
  then show ?case
proof (cases enat x + right I < enat nt)
  case False
  then have neg: enat x + right I \ge enat nt
  by auto
  have  $\bigwedge z. z \in \text{set } xs \implies \neg \text{enat } z + \text{right } I < \text{enat } nt$ 
proof -
  fix z
  assume z \in set xs
  then have enat z + right I \ge enat x + right I
  using Cons by auto
  with neg have enat z + right I \ge enat nt
  using dual_order.trans by blast
  then show  $\neg \text{enat } z + \text{right } I < \text{enat } nt$ 
  by auto
qed
  with False show ?thesis
  using filter_empty_conv by auto
qed auto
qed auto

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. enat t + right (args_ivl args) < 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 \ge cur
shows valid_mmuaux' args cur nt (shift_mmuaux args nt aux) auxlist  $\wedge$ 
  ( $\forall ts \in \text{set } (\text{lin\_ts\_mmuaux } (\text{shift\_mmuaux } args nt aux)). \neg \text{enat } ts + \text{right } (\text{args\_ivl } args) < nt$ )
proof -
  define I where I = args_ivl args
  define pos where pos = args_pos args
  have valid_folded: valid_mmuaux' args cur nt aux auxlist

```

```

using assms(1,2) valid_mmuaux'_mono unfolding valid_mmuaux_def by blast
obtain tp len tss maskL maskR a1_map a2_map done done_length where aux_def:
  aux = (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length)
by (cases aux) auto
note valid_before = valid_folded[unfolded aux_def]
define tss_list where tss_list =
  linearize (takeWhile_queue (λt. enat t + right I < enat nt) tss)
have tss_list_takeWhile: tss_list = takeWhile (λt. enat t + right I < enat nt) (linearize tss)
using tss_list_def unfolding takeWhile_queue_rep .
then obtain tss_list' where tss_list'_def: linearize tss = tss_list @ tss_list'
  tss_list' = dropWhile (λt. enat t + right I < enat nt) (linearize tss)
by auto
obtain tp' len' tss' maskL' maskR' a1_map' a2_map' done' done_length' where
  foldl_aux_def: (tp', tss', len', maskL', maskR', a1_map', a2_map',
    done', done_length') = foldl (λaux _. eval_step_mmuaux aux) aux tss_list
by (cases foldl (λaux _. eval_step_mmuaux aux) aux tss_list) auto
have lin_tss_aux: lin_ts_mmuaux aux = linearize tss
unfolding aux_def by auto
have take (length tss_list) (lin_ts_mmuaux aux) = tss_list
unfolding lin_tss_aux using tss_list'_def(1) by auto
then have valid_foldl: valid_mmuaux' args cur nt
  (foldl (λaux _. eval_step_mmuaux aux) aux tss_list) auxlist
  lin_ts_mmuaux (foldl (λaux _. eval_step_mmuaux aux) aux tss_list) = tss_list'
using valid_foldl_eval_step_mmuaux'[OF valid_before[folded aux_def], unfolded lin_tss_aux,
  OF tss_list'_def(1)] tss_list_takeWhile set_takeWhileD
unfolding lin_tss_aux I_def by fastforce+
have shift_mmuaux_eq: shift_mmuaux args nt aux = foldl (λaux _. eval_step_mmuaux aux) aux
tss_list
using tss_list_def unfolding aux_def I_def by auto
have  $\bigwedge ts. ts \in \text{set } tss\_list' \implies \neg \text{enat } ts + \text{right } (\text{args\_ivl } \text{args}) < nt$ 
using sorted_dropWhile_filter tss_list'_def(2) valid_before unfolding I_def by auto
then show ?thesis
using valid_foldl(1)[unfolded shift_mmuaux_eq[symmetric]]
unfolding valid_foldl(2)[unfolded shift_mmuaux_eq[symmetric]] by auto
qed

```

lift_definition *upd_set'* :: ('a, 'b) *mapping* \Rightarrow 'b \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a *set* \Rightarrow ('a, 'b) *mapping* **is**
 $\lambda m d f X a. (\text{if } a \in X \text{ then } (\text{case } \text{Mapping.lookup } m \ a \ \text{of } \text{Some } b \Rightarrow \text{Some } (f \ b) \mid \text{None} \Rightarrow \text{Some } d) \text{ else } \text{Mapping.lookup } m \ a) .$

lemma *upd_set'_lookup: Mapping.lookup (upd_set' m d f X) a = (if a ∈ X then*
(case Mapping.lookup m a of Some b \Rightarrow Some (f b) | None \Rightarrow Some d) else Mapping.lookup m a)
by (*simp add: Mapping.lookup.rep_eq upd_set'.rep_eq*)

lemma *upd_set'_keys: Mapping.keys (upd_set' m d f X) = Mapping.keys m \cup X*
by (*auto simp add: upd_set'_lookup intro!: Mapping_keys_intro*
dest!: Mapping_keys_dest split: option.splits)

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**
 $\lambda m d f X a. \text{case } \text{Mapping.lookup } m \ a \ \text{of } \text{Some } m' \Rightarrow \text{Some } (\text{upd_set}' \ m' \ d \ f \ \{b. (a, b) \in X\})$
 $\mid \text{None} \Rightarrow \text{if } a \in \text{fst } 'X \text{ then } \text{Some } (\text{upd_set}' \ \text{Mapping.empty } d \ f \ \{b. (a, b) \in X\}) \text{ else } \text{None} .$

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) ∈ X})
 $\mid \text{None} \Rightarrow \text{if } a \in \text{fst } 'X \text{ then } \text{Some } (\text{upd_set}' \ \text{Mapping.empty } d \ f \ \{b. (a, b) \in X\}) \text{ else } \text{None})$
by (*simp add: Mapping.lookup.abs_eq upd_nested.abs_eq*)

lemma *upd_nested_keys*: $\text{Mapping.keys } (\text{upd_nested } m \ d \ f \ X) = \text{Mapping.keys } m \cup \text{fst } 'X$
by (*auto simp add: upd_nested_lookup Domain.DomainI fst_eq_Domain intro!: Mapping_keys_intro dest!: Mapping_keys_dest split: option.splits*)

fun *add_new_mmuaux* :: $\text{args} \Rightarrow 'a \ \text{table} \Rightarrow 'a \ \text{table} \Rightarrow \text{ts} \Rightarrow 'a \ \text{mmuaux} \Rightarrow 'a \ \text{mmuaux}$ **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 = $\bigcup ((\lambda \text{as. case } \text{Mapping.lookup } a1_map \ (\text{proj_tuple } \text{maskL } \text{as}) \ \text{of } \text{None} \Rightarrow$
 (if $\neg \text{pos}$ then $\{(tp - len, \text{as})\}$ else $\{\}$)
 | *Some* *tp'* \Rightarrow if *pos* then $\{(\max (tp - len) \ tp', \text{as})\}$
 else $\{(\max (tp - len) \ (tp' + 1), \text{as})\}$ ' *rel2*) \cup (if *left I* = 0 then $\{tp\} \times \text{rel2}$ 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* ($\lambda _ . \text{as} \in \text{rel1}$)
 (*upd_set* *a1_map* ($\lambda _ . \text{tp}$) (*rel1* - *Mapping.keys* *a1_map*)) else *upd_set* *a1_map* ($\lambda _ . \text{tp}$) *rel1*);
tss = *append_queue* *nt* *tss* in
 (*tp* + 1, *tss*, *len* + 1, *maskL*, *maskR*, *a1_map*, *a2_map*, *done*, *done_length*))

lemma *fst_case*: $(\lambda x. \text{fst } (\text{case } x \ \text{of } (t, a1, a2) \Rightarrow (t, y \ t \ a1 \ a2, z \ t \ a1 \ a2))) = \text{fst}$
by *auto*

lemma *list_all2_in_setE*: $\text{list_all2 } P \ xs \ ys \Longrightarrow x \in \text{set } xs \Longrightarrow (\bigwedge y. y \in \text{set } ys \Longrightarrow P \ x \ y \Longrightarrow Q) \Longrightarrow Q$
by (*fastforce simp: list_all2_iff set_zip in_set_conv_nth*)

lemma *list_all2_zip*: $\text{list_all2 } (\lambda x \ y. \text{triple_eq_pair } x \ y \ f \ g) \ xs \ (\text{zip } ys \ zs) \Longrightarrow$
 ($\bigwedge y. y \in \text{set } ys \Longrightarrow Q \ y$) $\Longrightarrow x \in \text{set } xs \Longrightarrow Q \ (\text{fst } x)$
by (*auto simp: in_set_zip elim!: list_all2_in_setE triple_eq_pair.elims*)

lemma *list_appendE*: $xs = ys \ @ \ zs \Longrightarrow x \in \text{set } xs \Longrightarrow$
 ($x \in \text{set } ys \Longrightarrow P$) $\Longrightarrow (x \in \text{set } zs \Longrightarrow P) \Longrightarrow P$
by *auto*

lemma *take_takeWhile*: $n \leq \text{length } ys \Longrightarrow$
 ($\bigwedge y. y \in \text{set } (\text{take } n \ ys) \Longrightarrow P \ y$) \Longrightarrow
 ($\bigwedge y. y \in \text{set } (\text{drop } n \ ys) \Longrightarrow \neg P \ y$) \Longrightarrow
take *n* *ys* = *takeWhile* *P* *ys*

proof (*induction* *ys* *arbitrary: n*)
case *Nil*
then show ?*case* **by** *simp*
next
case (*Cons* *y* *ys*)
then show ?*case* **by** (*cases* *n*) *simp_all*
qed

lemma *valid_add_new_mmuaux*:
assumes *valid_before*: *valid_mmuaux* *args* *cur* *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 -
define *I* **where** $I = \text{args_ivl } \text{args}$
define *n* **where** $n = \text{args_n } \text{args}$
define *L* **where** $L = \text{args_L } \text{args}$
define *R* **where** $R = \text{args_R } \text{args}$

```

define pos where pos = args_pos args
have valid_folded: valid_mmuaux' args cur nt aux auxlist
  using assms(1,4) valid_mmuaux'_mono unfolding valid_mmuaux_def by blast
obtain tp len tss maskL maskR a1_map a2_map done done_length where shift_aux_def:
  shift_mmuaux args nt aux = (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length)
  by (cases shift_mmuaux args nt aux) auto
have valid_shift_aux: valid_mmuaux' args cur nt (tp, tss, len, maskL, maskR,
  a1_map, a2_map, done, done_length) auxlist
   $\wedge$  ts. ts  $\in$  set (linearize tss)  $\implies$   $\neg$ enat ts + right (args_ivl args) < enat nt
  using valid_shift_mmuaux'[OF assms(1)[unfolded valid_mmuaux_def] assms(4)]
  unfolding shift_aux_def by auto
define new_tstp where new_tstp = (if left I = 0 then Inr tp else Inl (nt - (left I - 1)))
have new_tstp_lt_isl: tstp_lt new_tstp (nt - (left I - 1)) (tp + 1)
  isl new_tstp  $\iff$  left I > 0
  by (auto simp add: new_tstp_def tstp_lt_def)
define tmp where tmp =  $\bigcup$ (( $\lambda$ as. case Mapping.lookup a1_map (proj_tuple maskL as) of None  $\implies$ 
  (if  $\neg$ pos then {(tp - len, as)} else {}))
  | Some tp'  $\implies$  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 {})
have a1_map_lookup:  $\wedge$ as tp'. Mapping.lookup a1_map as = Some tp'  $\implies$  tp' < tp
  using valid_shift_aux(1) Mapping_keys_intro by force
then have fst_tmp:  $\wedge$ tp'. tp'  $\in$  fst ' tmp  $\implies$  tp - len  $\leq$  tp'  $\wedge$  tp' < tp + 1
  unfolding tmp_def by (auto simp add: less_SucI split: option.splits if_splits)
have snd_tmp:  $\wedge$ tp'. table n R (snd ' tmp)
  using tabs(2) unfolding tmp_def n_def R_def
  by (auto simp add: table_def split: if_splits option.splits)
define a2_map' where a2_map' = Mapping.update (tp + 1) Mapping.empty
  (upd_nested a2_map new_tstp (max_tstp new_tstp) tmp)
define a1_map' where 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)
define tss' where tss' = append_queue nt tss
have add_new_mmuaux_eq: add_new_mmuaux args rel1 rel2 nt aux = (tp + 1, tss', len + 1,
  maskL, maskR, a1_map', a2_map', done, done_length)
  using shift_aux_def new_tstp_def tmp_def a2_map'_def a1_map'_def tss'_def
  unfolding I_def pos_def
  by (auto simp only: add_new_mmuaux.simps Let_def)
have update_until_eq: update_until args rel1 rel2 nt auxlist =
  (map ( $\lambda$ x. case x of (t, a1, a2)  $\implies$  (t, if pos then join a1 True rel1 else a1  $\cup$  rel1,
  if mem (nt - t) I then a2  $\cup$  join rel2 pos a1 else a2)) auxlist) @
  [(nt, rel1, if left I = 0 then rel2 else empty_table)]
  unfolding update_until_def I_def pos_def by simp
have len_done_auxlist: length done  $\leq$  length auxlist
  using valid_shift_aux by auto
have auxlist_split: auxlist = take (length done) auxlist @ drop (length done) auxlist
  using len_done_auxlist by auto
have lin_tss': linearize tss' = linearize tss @ [nt]
  unfolding tss'_def append_queue_rep by (rule refl)
have len_lin_tss': length (linearize tss') = len + 1
  unfolding lin_tss' using valid_shift_aux by auto
have tmp: sorted (linearize tss)  $\wedge$ t. t  $\in$  set (linearize tss)  $\implies$  t  $\leq$  cur
  using valid_shift_aux by auto
have sorted_lin_tss': sorted (linearize tss')
  unfolding lin_tss' using tmp(1) le_trans[OF _ assms(4), OF tmp(2)]
  by (simp add: sorted_append)
have in_lin_tss:  $\wedge$ t. t  $\in$  set (linearize tss)  $\implies$ 
  t  $\leq$  cur  $\wedge$  enat cur  $\leq$  enat t + right I
  using valid_shift_aux(1) unfolding I_def by auto
then have set_lin_tss':  $\forall$ t  $\in$  set (linearize tss'). t  $\leq$  nt  $\wedge$  enat nt  $\leq$  enat t + right I

```

```

unfolding lin_tss' I_def using le_trans[OF _ assms(4)] valid_shift_aux(2)
by (auto simp add: not_less)
have a1_map'_keys: Mapping.keys a1_map'  $\subseteq$  Mapping.keys a1_map  $\cup$  rel1
unfolding a1_map'_def using Mapping.keys_filter Mapping_upd_set_keys
by (auto simp add: Mapping_upd_set_keys split: if_splits dest: Mapping_keys_filterD)
then have tab_a1_map'_keys: table n L (Mapping.keys a1_map')
using valid_shift_aux(1) tabs(1) by (auto simp add: table_def n_def L_def)
have a2_map_keys: Mapping.keys a2_map = {tp - len..tp}
using valid_shift_aux by auto
have a2_map'_keys: Mapping.keys a2_map' = {tp - len..tp + 1}
unfolding a2_map'_def Mapping.keys_update upd_nested_keys a2_map_keys using fst_tmp
by fastforce
then have a2_map'_keys': Mapping.keys a2_map' = {tp + 1 - (len + 1)..tp + 1}
by auto
have len_upd_until: length done + (len + 1) = length (update_until args rel1 rel2 nt auxlist)
using valid_shift_aux unfolding update_until_eq by auto
have set_take_auxlist:  $\bigwedge x. x \in \text{set } (\text{take } (\text{length done}) \text{ auxlist}) \implies \text{check\_before } I \text{ nt } x$ 
using valid_shift_aux unfolding I_def by auto
have list_all2_triple: list_all2 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map } \text{pos } tp' \text{ a1\_map})$ )
( $\lambda ts' tp'. \text{filter\_a2\_map } I \text{ ts' } tp' \text{ a2\_map}$ ) (drop (length done) auxlist)
(zip (linearize tss) [tp - len.. $tp$ ])
using valid_shift_aux unfolding I_def pos_def by auto
have set_drop_auxlist:  $\bigwedge x. x \in \text{set } (\text{drop } (\text{length done}) \text{ auxlist}) \implies \neg \text{check\_before } I \text{ nt } x$ 
using valid_shift_aux(2)[OF list_all2_zip[OF list_all2_triple,
of  $\lambda y. y \in \text{set } (\text{linearize } tss)$ ]]
unfolding I_def by auto
have length_done_auxlist: length done  $\leq$  length auxlist
using valid_shift_aux by auto
have take_auxlist_takeWhile: take (length done) auxlist = takeWhile (check_before I nt) auxlist
using take_takeWhile[OF length_done_auxlist set_take_auxlist set_drop_auxlist] .
have length_done = length (takeWhile (check_before I nt) auxlist)
by (metis (no_types) add_diff_cancel_right' auxlist_split diff_diff_cancel
length_append length_done_auxlist length_drop take_auxlist_takeWhile)
then have set_take_auxlist':  $\bigwedge x. x \in \text{set } (\text{take } (\text{length done})$ 
(update_until args rel1 rel2 nt auxlist))  $\implies \text{check\_before } I \text{ nt } x$ 
by (metis I_def length_map map_proj_thd update_until set_takeWhileD takeWhile_eq_take)
have rev_done: rev done = map proj_thd (take (length done) auxlist)
using valid_shift_aux by auto
moreover have ... = map proj_thd (takeWhile (check_before I nt)
(update_until args rel1 rel2 nt auxlist))
by (simp add: take_auxlist_takeWhile map_proj_thd update_until I_def)
finally have rev_done': rev done = map proj_thd (take (length done)
(update_until args rel1 rel2 nt auxlist))
by (metis length_map length_rev takeWhile_eq_take)
have map_fst_auxlist_take:  $\bigwedge t. t \in \text{set } (\text{map } \text{fst } (\text{take } (\text{length done}) \text{ auxlist})) \implies t \leq \text{nt}$ 
using set_take_auxlist
by auto (meson add_increasing2 enat_ord_simps(1) le_cases not_less zero_le)
have map_fst_auxlist_drop:  $\bigwedge t. t \in \text{set } (\text{map } \text{fst } (\text{drop } (\text{length done}) \text{ auxlist})) \implies t \leq \text{nt}$ 
using in_lin_tss[OF list_all2_zip[OF list_all2_triple, of  $\lambda y. y \in \text{set } (\text{linearize } tss)$ ]]
assms(4) dual_order.trans by auto blast
have set_drop_auxlist_cong:  $\bigwedge x t a1 a2. x \in \text{set } (\text{drop } (\text{length done}) \text{ auxlist}) \implies$ 
 $x = (t, a1, a2) \implies \text{mem } (\text{nt} - t) I \iff \text{left } I \leq \text{nt} - t$ 
proof -
fix x t a1 a2
assume  $x \in \text{set } (\text{drop } (\text{length done}) \text{ auxlist})$   $x = (t, a1, a2)$ 
then have enat t + right I  $\geq$  enat nt
using set_drop_auxlist not_less
by auto blast

```

```

then have right I ≥ enat (nt - t)
  by (cases right I) auto
then show mem (nt - t) I ↔ left I ≤ nt - t
  by auto
qed
have sorted_fst_auxlist: sorted (map fst auxlist)
  using valid_shift_aux by auto
have set_map_fst_auxlist: ∧t. t ∈ set (map fst auxlist) ⇒ t ≤ nt
  using arg_cong[OF auxlist_split, of map fst, unfolded map_append] map_fst_auxlist_take
  map_fst_auxlist_drop by auto
have lookup_a1_map_keys: ∧xs tp'. Mapping.lookup a1_map xs = Some tp' ⇒ tp' < tp
  using valid_shift_aux Mapping_keys_intro by force
have lookup_a1_map_keys': ∀xs ∈ Mapping.keys a1_map'.
  case Mapping.lookup a1_map' xs of Some tp' ⇒ tp' < tp + 1
  using lookup_a1_map_keys unfolding a1_map'_def
  by (auto simp add: Mapping.lookup_filter Mapping_lookup_upd_set Mapping_upd_set_keys
    split: option.splits dest: Mapping_keys_dest) fastforce+
have sorted_upd_until: sorted (map fst (update_until args rel1 rel2 nt auxlist))
  using sorted_fst_auxlist set_map_fst_auxlist
  unfolding update_until_eq
  by (auto simp add: sorted_append comp_def fst_case)
have lookup_a2_map: ∧tp' m. Mapping.lookup a2_map tp' = Some m ⇒
  table n R (Mapping.keys m) ∧ (∀xs ∈ Mapping.keys m. case Mapping.lookup m xs of Some tstp ⇒
  tstp_lt tstp (cur - (left I - 1)) tp ∧ (isl tstp ↔ left I > 0))
  using valid_shift_aux(1) Mapping_keys_intro unfolding I_def n_def R_def by force
then have lookup_a2_map': ∧tp' m xs tstp. Mapping.lookup a2_map tp' = Some m ⇒
  Mapping.lookup m xs = Some tstp ⇒ tstp_lt tstp (nt - (left I - 1)) tp ∧
  isl tstp = (0 < left I)
  using Mapping_keys_intro assms(4) by (force simp add: tstp_lt_def split: sum.splits)
have lookup_a2_map'_keys: ∀tp' ∈ Mapping.keys a2_map'.
  case Mapping.lookup a2_map' tp' of Some m ⇒ table n R (Mapping.keys m) ∧
  (∀xs ∈ Mapping.keys m. case Mapping.lookup m xs of Some tstp ⇒
  tstp_lt tstp (nt - (left I - 1)) (tp + 1) ∧ isl tstp = (0 < left I))
proof (rule ballI)
  fix tp'
  assume tp'_assm: tp' ∈ Mapping.keys a2_map'
  then obtain m' where m'_def: Mapping.lookup a2_map' tp' = Some m'
    by (auto dest: Mapping_keys_dest)
  have table n R (Mapping.keys m') ∧
    (∀xs ∈ Mapping.keys m'. case Mapping.lookup m' xs of Some tstp ⇒
    tstp_lt tstp (nt - (left I - 1)) (tp + 1) ∧ isl tstp = (0 < left I))
  proof (cases tp' = tp + 1)
    case True
    show ?thesis
      using m'_def unfolding a2_map'_def True Mapping.lookup_update
      by (auto simp add: table_def)
  next
    case False
    then have tp'_in: tp' ∈ Mapping.keys a2_map
      using tp'_assm unfolding a2_map_keys a2_map'_keys by auto
    then obtain m where m_def: Mapping.lookup a2_map tp' = Some m
      by (auto dest: Mapping_keys_dest)
    have m'_alt: m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp', b) ∈ tmp}
      using m_def m'_def unfolding a2_map'_def Mapping.lookup_update_neq[OF False[symmetric]]
      upd_nested_lookup
      by auto
    have table n R (Mapping.keys m')
      using lookup_a2_map[OF m_def] snd_tmp unfolding m'_alt upd_set'_keys

```

```

    by (auto simp add: table_def)
  moreover have  $\forall xs \in \text{Mapping.keys } m'. \text{case Mapping.lookup } m' \text{ } xs \text{ of Some } tstp \Rightarrow$ 
     $tstp\_lt \ tstp \ (nt - (\text{left } I - 1)) \ (tp + 1) \wedge \text{isl } tstp = (0 < \text{left } I)$ 
  proof (rule ballI)
    fix xs
    assume  $xs\_assm: xs \in \text{Mapping.keys } m'$ 
    then obtain  $tstp$  where  $tstp\_def: \text{Mapping.lookup } m' \ xs = \text{Some } tstp$ 
      by (auto dest: Mapping_keys_dest)
    have  $tstp\_lt \ tstp \ (nt - (\text{left } I - 1)) \ (tp + 1) \wedge \text{isl } tstp = (0 < \text{left } I)$ 
    proof (cases Mapping.lookup m xs)
      case None
      then show ?thesis
        using  $tstp\_def[\text{unfolded } m'\_alt \ upd\_set'\_lookup] \ new\_tstp\_lt\_isl$ 
        by (auto split: if_splits)
    next
      case (Some  $tstp'$ )
      show ?thesis
      proof (cases  $xs \in \{b. (tp', b) \in tmp\}$ )
        case True
        then have  $tstp\_eq: tstp = \max\_tstp \ new\_tstp \ tstp'$ 
          using  $tstp\_def[\text{unfolded } m'\_alt \ upd\_set'\_lookup] \ Some$ 
          by auto
        show ?thesis
          using  $lookup\_a2\_map'[\text{OF } m\_def \ Some] \ new\_tstp\_lt\_isl$ 
          by (auto simp add:  $tstp\_lt\_def \ tstp\_eq$  split: sum.splits)
      next
        case False
        then show ?thesis
          using  $tstp\_def[\text{unfolded } m'\_alt \ upd\_set'\_lookup] \ lookup\_a2\_map'[\text{OF } m\_def \ Some] \ Some$ 
          by (auto simp add:  $tstp\_lt\_def$  split: sum.splits)
      qed
    qed
  ultimately show ?thesis
    by auto
  qed
  then show  $\text{case Mapping.lookup } m' \ xs \text{ of Some } tstp \Rightarrow$ 
     $tstp\_lt \ tstp \ (nt - (\text{left } I - 1)) \ (tp + 1) \wedge \text{isl } tstp = (0 < \text{left } I)$ 
    using  $tstp\_def$  by auto
  qed
  ultimately show ?thesis
    by auto
  qed
  then show  $\text{case Mapping.lookup } a2\_map' \ tp' \text{ of Some } m \Rightarrow \text{table } n \ R \ (\text{Mapping.keys } m) \wedge$ 
     $(\forall xs \in \text{Mapping.keys } m. \text{case Mapping.lookup } m \ xs \text{ of Some } tstp \Rightarrow$ 
     $tstp\_lt \ tstp \ (nt - (\text{left } I - 1)) \ (tp + 1) \wedge \text{isl } tstp = (0 < \text{left } I))$ 
    using  $m'\_def$  by auto
  qed
  have  $tp\_upt\_Suc: [tp + 1 - (\text{len} + 1)..<tp + 1] = [tp - \text{len}..<tp] \ @ \ [tp]$ 
    using  $upt\_Suc$  by auto
  have  $map\_eq: \text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } pos \text{ then } join \ a1 \ True \ rel1 \ \text{else } a1 \cup \ rel1,$ 
     $\text{if } mem \ (nt - t) \ I \ \text{then } a2 \cup \ join \ rel2 \ pos \ a1 \ \text{else } a2)) \ (\text{drop } (\text{length } done) \ auxlist) =$ 
     $\text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } pos \ \text{then } join \ a1 \ True \ rel1 \ \text{else } a1 \cup \ rel1,$ 
     $\text{if } \text{left } I \leq nt - t \ \text{then } a2 \cup \ join \ rel2 \ pos \ a1 \ \text{else } a2)) \ (\text{drop } (\text{length } done) \ auxlist)$ 
    using  $set\_drop\_auxlist\_cong$  by auto
  have  $\text{drop } (\text{length } done) \ (\text{update\_until } args \ rel1 \ rel2 \ nt \ auxlist) =$ 
     $\text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } pos \ \text{then } join \ a1 \ True \ rel1 \ \text{else } a1 \cup \ rel1,$ 
     $\text{if } mem \ (nt - t) \ I \ \text{then } a2 \cup \ join \ rel2 \ pos \ a1 \ \text{else } a2)) \ (\text{drop } (\text{length } done) \ auxlist) \ @$ 
     $[(nt, \ rel1, \ \text{if } \text{left } I = 0 \ \text{then } rel2 \ \text{else } \text{empty\_table})]$ 
    unfolding  $\text{update\_until\_eq}$  using  $\text{len\_done\_auxlist } \text{drop\_map}$  by auto
  note  $\text{drop\_update\_until} = \text{this}[\text{unfolded } map\_eq]$ 
  have  $\text{list\_all2\_old}: \text{list\_all2 } (\lambda x \ y. \text{triple\_eq\_pair } x \ y \ (\lambda tp'. \text{filter\_a1\_map } pos \ tp' \ a1\_map'))$ 

```

```

( $\lambda ts' tp'. \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map'$ )
( $\text{map } (\lambda(t, a1, a2). (t, \text{if pos then join a1 True rel1 else a1 } \cup \text{ rel1,}$ 
   $\text{if left } I \leq nt - t \text{ then } a2 \cup \text{ join rel2 pos a1 else } a2)) (\text{drop } (\text{length done}) \text{ auxlist}))$ )
( $\text{zip } (\text{linearize tss}) [tp - \text{len}..<tp]$ )
unfolding list_all2_map1
using list_all2_triple
proof (rule list.rel_mono_strong)
fix tri pair
assume tri_pair_in:  $tri \in \text{set } (\text{drop } (\text{length done}) \text{ auxlist})$ 
   $pair \in \text{set } (\text{zip } (\text{linearize tss}) [tp - \text{len}..<tp])$ 
obtain  $t \ a1 \ a2$  where tri_def:  $tri = (t, a1, a2)$ 
  by (cases tri) auto
obtain  $ts' \ tp'$  where pair_def:  $pair = (ts', tp')$ 
  by (cases pair) auto
assume triple_eq_pair tri pair ( $\lambda tp'. \text{filter\_a1\_map pos } tp' \text{ } a1\_map$ )
  ( $\lambda ts' tp'. \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map$ )
then have eqs:  $t = ts' \ a1 = \text{filter\_a1\_map pos } tp' \text{ } a1\_map$ 
   $a2 = \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map$ 
unfolding tri_def pair_def by auto
have tp'_ge:  $tp' \geq tp - \text{len}$ 
  using tri_pair_in(2) unfolding pair_def
  by (auto elim: in_set_zipE)
have tp'_lt_tp:  $tp' < tp$ 
  using tri_pair_in(2) unfolding pair_def
  by (auto elim: in_set_zipE)
have ts'_in_lin_tss:  $ts' \in \text{set } (\text{linearize tss})$ 
  using tri_pair_in(2) unfolding pair_def
  by (auto elim: in_set_zipE)
then have ts'_nt:  $ts' \leq nt$ 
  using valid_shift_aux(1) assms(4) by auto
then have t_nt:  $t \leq nt$ 
  unfolding eqs(1) .
have table n L (Mapping.keys a1_map)
  using valid_shift_aux unfolding n_def L_def by auto
then have a1_tab: table n L a1
  unfolding eqs(2) filter_a1_map_def by (auto simp add: table_def)
note tabR = tabs(2)[unfolded n_def[symmetric] R_def[symmetric]]
have join_rel2_assms:  $L \subseteq R \ \text{maskL} = \text{join\_mask } n \ L$ 
  using valid_shift_aux unfolding n_def L_def R_def by auto
have join_rel2_eq:  $\text{join rel2 pos } a1 = \{xs \in \text{rel2}. \text{proj\_tuple\_in\_join pos maskL } xs \ a1\}$ 
  using join_sub[OF join_rel2_assms(1) a1_tab tabR] join_rel2_assms(2) by auto
have filter_sub_a2:  $\bigwedge xs \ m' \ tp'' \ tstp. tp'' \leq tp' \implies$ 
   $\text{Mapping.lookup } a2\_map' \ tp'' = \text{Some } m' \implies \text{Mapping.lookup } m' \ xs = \text{Some } tstp \implies$ 
   $ts\_tp\_lt' \ ts' \ tp' \ tstp \implies (tstp = \text{new\_tstp} \implies \text{False}) \implies$ 
   $xs \in \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map' \implies xs \in a2$ 
proof –
  fix  $xs \ m' \ tp'' \ tstp$ 
assume m'_def:  $tp'' \leq tp' \ \text{Mapping.lookup } a2\_map' \ tp'' = \text{Some } m'$ 
   $\text{Mapping.lookup } m' \ xs = \text{Some } tstp \ ts\_tp\_lt' \ ts' \ tp' \ tstp$ 
have tp''_neq:  $tp + 1 \neq tp''$ 
  using le_less_trans[OF m'_def(1) tp'_lt_tp] by auto
assume new_tstp_False:  $tstp = \text{new\_tstp} \implies \text{False}$ 
show  $xs \in a2$ 
proof (cases Mapping.lookup a2_map tp'')
  case None
then have m'_alt:  $m' = \text{upd\_set}' \ \text{Mapping.empty } \text{new\_tstp} \ (\text{max\_tstp } \text{new\_tstp})$ 
   $\{b. (tp'', b) \in \text{tmp}\}$ 
  using m'_def(2)[unfolded a2_map'_def Mapping.lookup_update_neq[OF tp''_neq]

```

```

    upd_nested_lookup] by (auto split: option.splits if_splits)
  then show ?thesis
    using new_tstp_False m'_def(3)[unfolded m'_alt upd_set'_lookup Mapping.lookup_empty]
    by (auto split: if_splits)
next
case (Some m)
then have m'_alt: m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp'', b) ∈ tmp}
  using m'_def(2)[unfolded a2_map'_def Mapping.lookup_update_neq[OF tp''_neq]
    upd_nested_lookup] by (auto split: option.splits if_splits)
note lookup_m = Some
show ?thesis
proof (cases Mapping.lookup m xs)
  case None
  then show ?thesis
    using new_tstp_False m'_def(3)[unfolded m'_alt upd_set'_lookup]
    by (auto split: if_splits)
next
case (Some tstp')
have tstp_ok: tstp = tstp' ⇒ xs ∈ a2
  using eqs(3) lookup_m Some m'_def unfolding filter_a2_map_def by auto
show ?thesis
proof (cases xs ∈ {b. (tp'', b) ∈ tmp})
  case True
  then have tstp_eq: tstp = max_tstp new_tstp tstp'
    using m'_def(3)[unfolded m'_alt upd_set'_lookup Some] by auto
  show ?thesis
    using lookup_a2_map'[OF lookup_m Some] new_tstp_lt_isl(2)
    tstp_eq new_tstp_False tstp_ok
    by (auto intro: max_tstpE[of new_tstp tstp'])
next
case False
then have tstp = tstp'
  using m'_def(3)[unfolded m'_alt upd_set'_lookup Some] by auto
then show ?thesis
  using tstp_ok by auto
qed
qed
qed
qed
have a2_sub_filter: a2 ⊆ filter_a2_map I ts' tp' a2_map'
proof (rule subsetI)
  fix xs
  assume xs_in: xs ∈ a2
  then obtain tp'' m tstp where m_def: tp'' ≤ tp' Mapping.lookup a2_map tp'' = Some m
    Mapping.lookup m xs = Some tstp ts_tp_lt' ts' tp' tstp
  using eqs(3)[unfolded filter_a2_map_def] by (auto split: option.splits)
  have tp''_in: tp'' ∈ {tp - len..tp}
  using m_def(2) a2_map_keys by (auto intro!: Mapping_keys_intro)
  then obtain m' where m'_def: Mapping.lookup a2_map' tp'' = Some m'
  using a2_map'_keys
  by (metis Mapping_keys_dest One_nat_def add_Suc_right add_diff_cancel_right'
    atLeastatMost_subset_iff diff_zero le_eq_less_or_eq le_less_Suc_eq subsetD)
  have tp''_neq: tp + 1 ≠ tp''
  using m_def(1) tp'_lt_tp by auto
  have m'_alt: m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp'', b) ∈ tmp}
  using m'_def[unfolded a2_map'_def Mapping.lookup_update_neq[OF tp''_neq] m_def(2)
    upd_nested_lookup] by (auto split: option.splits if_splits)
  show xs ∈ filter_a2_map I ts' tp' a2_map'

```

```

proof (cases xs ∈ {b. (tp'', b) ∈ tmp})
  case True
  then have Mapping.lookup m' xs = Some (max_tstp new_tstp tstp)
    unfolding m'_alt_upd_set'_lookup m_def(3) by auto
  moreover have ts_tp_lt' ts' tp' (max_tstp new_tstp tstp)
    using new_tstp_lt_isl(2) lookup_a2_map'[OF m_def(2,3)]
    by (auto intro: max_tstp_intro''''[OF m_def(4)])
  ultimately show ?thesis
    unfolding filter_a2_map_def using m_def(1) m'_def m_def(4) by auto
  next
  case False
  then have Mapping.lookup m' xs = Some tstp
    unfolding m'_alt_upd_set'_lookup m_def(3) by auto
  then show ?thesis
    unfolding filter_a2_map_def using m_def(1) m'_def m_def by auto
  qed
qed
have pos ⇒ filter_a1_map pos tp' a1_map' = join a1 True rel1
proof -
  assume pos: pos
  note tabL = tabs(1)[unfolded n_def[symmetric] L_def[symmetric]]
  have join_eq: join a1 True rel1 = a1 ∩ rel1
    using join_eq[OF tabL a1_tab] by auto
  show filter_a1_map pos tp' a1_map' = join a1 True rel1
    using eqs(2) pos tp'_lt_tp unfolding filter_a1_map_def a1_map'_def join_eq
    by (auto simp add: Mapping.lookup_filter Mapping_lookup_upd_set split: if_splits option.splits
      intro: Mapping_keys_intro dest: Mapping_keys_dest Mapping_keys_filterD)
  qed
moreover have ¬pos ⇒ filter_a1_map pos tp' a1_map' = a1 ∪ rel1
  using eqs(2) tp'_lt_tp unfolding filter_a1_map_def a1_map'_def
  by (auto simp add: Mapping.lookup_filter Mapping_lookup_upd_set intro: Mapping_keys_intro
    dest: Mapping_keys_filterD Mapping_keys_dest split: option.splits)
moreover have left I ≤ nt - t ⇒ filter_a2_map I ts' tp' a2_map' = a2 ∪ join rel2 pos a1
proof (rule set_eqI, rule iffI)
  fix xs
  assume in_int: left I ≤ nt - t
  assume xs_in: xs ∈ filter_a2_map I ts' tp' a2_map'
  then obtain m' tp'' tstp where m'_def: tp'' ≤ tp' Mapping.lookup a2_map' tp'' = Some m'
    Mapping.lookup m' xs = Some tstp ts_tp_lt' ts' tp' tstp
    unfolding filter_a2_map_def by (fastforce split: option.splits)
  show xs ∈ a2 ∪ join rel2 pos a1
  proof (cases tstp = new_tstp)
    case True
    note tstp_new_tstp = True
    have tp''_neq: tp + 1 ≠ tp''
      using m'_def(1) tp'_lt_tp by auto
    have tp''_in: tp'' ∈ {tp - len..tp}
      using m'_def(1,2) tp'_lt_tp a2_map'_keys
      by (auto intro!: Mapping_keys_intro)
    obtain m where m_def: Mapping.lookup a2_map tp'' = Some m
      m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp'', b) ∈ tmp}
    using m'_def(2)[unfolded a2_map'_def Mapping.lookup_update_neq[OF tp''_neq]
      upd_nested_lookup] tp''_in a2_map_keys
    by (fastforce dest: Mapping_keys_dest split: option.splits if_splits)
    show ?thesis
  proof (cases Mapping.lookup m xs = Some new_tstp)
    case True
    then show ?thesis

```

```

    using eqs(3) m'_def(1) m_def(1) m'_def tstp_new_tstp
    unfolding filter_a2_map_def by auto
next
case False
then have xs_in_snd_tmp: xs ∈ {b. (tp'', b) ∈ tmp}
    using m'_def(3)[unfolded m_def(2) upd_set'_lookup True]
    by (auto split: if_splits)
then have xs_in_rel2: xs ∈ rel2
    unfolding tmp_def
    by (auto split: if_splits option.splits)
show ?thesis
proof (cases pos)
case True
obtain tp''' where tp'''_def: Mapping.lookup a1_map (proj_tuple maskL xs) = Some tp'''
    if pos then tp'' = max (tp - len) tp''' else tp'' = max (tp - len) (tp''' + 1)
    using xs_in_snd_tmp m'_def(1) tp'_lt_tp True
    unfolding tmp_def by (auto split: option.splits if_splits)
have proj_tuple maskL xs ∈ a1
    using eqs(2)[unfolded filter_a1_map_def] True m'_def(1) tp'''_def
    by (auto intro: Mapping_keys_intro)
then show ?thesis
    using True xs_in_rel2 unfolding proj_tuple_in_join_def join_rel2_eq by auto
next
case False
show ?thesis
proof (cases Mapping.lookup a1_map (proj_tuple maskL xs))
case None
then show ?thesis
    using xs_in_rel2 False eqs(2)[unfolded filter_a1_map_def]
    unfolding proj_tuple_in_join_def join_rel2_eq
    by (auto dest: Mapping_keys_dest)
next
case (Some tp''')
then have tp'' = max (tp - len) (tp''' + 1)
    using xs_in_snd_tmp m'_def(1) tp'_lt_tp False
    unfolding tmp_def by (auto split: option.splits if_splits)
then have tp''' < tp'
    using m'_def(1) by auto
then have proj_tuple maskL xs ∉ a1
    using eqs(2)[unfolded filter_a1_map_def] True m'_def(1) Some False
    by (auto intro: Mapping_keys_intro)
then show ?thesis
    using xs_in_rel2 False unfolding proj_tuple_in_join_def join_rel2_eq by auto
qed
qed
qed
next
case False
then show ?thesis
    using filter_sub_a2[OF m'_def xs_in] by auto
qed
next
fix xs
assume in_int: left I ≤ nt - t
assume xs_in: xs ∈ a2 ∪ join_rel2 pos a1
then have xs ∈ a2 ∪ (join_rel2 pos a1 - a2)
    by auto
then show xs ∈ filter_a2_map I ts' tp' a2_map'

```

```

proof (rule UnE)
  assume  $xs \in a2$ 
  then show  $xs \in \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map'$ 
    using  $a2\_sub\_filter$  by auto
next
  assume  $xs \in \text{join\_rel2 } pos \text{ } a1 - a2$ 
  then have  $xs\_props: xs \in \text{rel2 } xs \notin a2 \text{ } \text{proj\_tuple\_in\_join } pos \text{ } \text{maskL } xs \text{ } a1$ 
    unfolding  $\text{join\_rel2\_eq}$  by auto
  have  $ts\_tp\_lt'\_new\_tstp: ts\_tp\_lt' \text{ } ts' \text{ } tp' \text{ } new\_tstp$ 
    using  $tp'\_lt\_tp \text{ } in\_int \text{ } t\_nt \text{ } eqs(1)$  unfolding  $new\_tstp\_def$ 
    by ( $auto \text{ simp add: } ts\_tp\_lt'\_def$ )
  show  $xs \in \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map'$ 
proof (cases  $pos$ )
  case True
  then obtain  $tp'''$  where  $tp'''\_def: tp''' \leq tp'$ 
     $\text{Mapping.lookup } a1\_map \text{ } (\text{proj\_tuple } \text{maskL } xs) = \text{Some } tp'''$ 
    using  $eqs(2)[\text{unfolded } \text{filter\_a1\_map\_def}] \text{ } xs\_props(3)[\text{unfolded } \text{proj\_tuple\_in\_join\_def}]$ 
    by ( $auto \text{ dest: } \text{Mapping\_keys\_dest}$ )
  define  $wtp$  where  $wtp \equiv \max (tp - len) \text{ } tp'''$ 
  have  $wtp\_xs\_in: (wtp, xs) \in tmp$ 
    unfolding  $wtp\_def$  using  $tp'''\_def \text{ } tmp\_def \text{ } xs\_props(1)$  True by fastforce
  have  $wtp\_le: wtp \leq tp'$ 
    using  $tp'''\_def(1) \text{ } tp'\_ge$  unfolding  $wtp\_def$  by auto
  have  $wtp\_in: wtp \in \{tp - len..tp\}$ 
    using  $tp'''\_def(1) \text{ } tp'\_lt\_tp$  unfolding  $wtp\_def$  by auto
  have  $wtp\_neq: tp + 1 \neq wtp$ 
    using  $wtp\_in$  by auto
  obtain  $m$  where  $m\_def: \text{Mapping.lookup } a2\_map \text{ } wtp = \text{Some } m$ 
    using  $wtp\_in \text{ } a2\_map\_keys \text{ } \text{Mapping\_keys\_dest}$  by fastforce
  obtain  $m'$  where  $m'\_def: \text{Mapping.lookup } a2\_map' \text{ } wtp = \text{Some } m'$ 
    using  $wtp\_in \text{ } a2\_map'\_keys \text{ } \text{Mapping\_keys\_dest}$  by fastforce
  have  $m'\_alt: m' = \text{upd\_set}' m \text{ } new\_tstp \text{ } (\max\_tstp \text{ } new\_tstp) \{b. (wtp, b) \in tmp\}$ 
    using  $m'\_def[\text{unfolded } a2\_map'\_def] \text{ } \text{Mapping.lookup\_update\_neq}[OF \text{ } wtp\_neq]$ 
     $\text{upd\_nested\_lookup } m\_def]$  by auto
  show  $?thesis$ 
proof (cases  $\text{Mapping.lookup } m \text{ } xs$ )
  case None
  have  $\text{Mapping.lookup } m' \text{ } xs = \text{Some } new\_tstp$ 
    using  $wtp\_xs\_in$  unfolding  $m'\_alt \text{ } \text{upd\_set}'\_lookup$  None by auto
  then show  $?thesis$ 
    unfolding  $\text{filter\_a2\_map\_def}$  using  $wtp\_le \text{ } m'\_def \text{ } ts\_tp\_lt'\_new\_tstp$  by auto
next
  case ( $\text{Some } tstp'$ )
  have  $\text{Mapping.lookup } m' \text{ } xs = \text{Some } (\max\_tstp \text{ } new\_tstp \text{ } tstp')$ 
    using  $wtp\_xs\_in$  unfolding  $m'\_alt \text{ } \text{upd\_set}'\_lookup$  Some by auto
  moreover have  $ts\_tp\_lt' \text{ } ts' \text{ } tp' \text{ } (\max\_tstp \text{ } new\_tstp \text{ } tstp')$ 
using  $\max\_tstp\_intro'''' \text{ } ts\_tp\_lt'\_new\_tstp \text{ } \text{lookup\_a2\_map}'[OF \text{ } m\_def \text{ } \text{Some}] \text{ } new\_tstp\_lt\_isl$ 
by auto
  ultimately show  $?thesis$ 
    using  $\text{lookup\_a2\_map}'[OF \text{ } m\_def \text{ } \text{Some}] \text{ } wtp\_le \text{ } m'\_def$ 
    unfolding  $\text{filter\_a2\_map\_def}$  by auto
qed
next
  case False
  show  $?thesis$ 
proof (cases  $\text{Mapping.lookup } a1\_map \text{ } (\text{proj\_tuple } \text{maskL } xs)$ )
  case None
  then have  $in\_tmp: (tp - len, xs) \in tmp$ 

```

```

    using tmp_def False xs_props(1) by fastforce
  obtain m where m_def: Mapping.lookup a2_map (tp - len) = Some m
    using a2_map_keys by (fastforce dest: Mapping_keys_dest)
  obtain m' where m'_def: Mapping.lookup a2_map' (tp - len) = Some m'
    using a2_map'_keys by (fastforce dest: Mapping_keys_dest)
  have tp_neg: tp + 1 ≠ tp - len
    by auto
  have m'_alt: m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp - len, b) ∈ tmp}
    using m'_def[unfolded a2_map'_def Mapping.lookup_update_neq[OF tp_neg]
      upd_nested_lookup m_def] by auto
  show ?thesis
  proof (cases Mapping.lookup m xs)
    case None
    have Mapping.lookup m' xs = Some new_tstp
      unfolding m'_alt upd_set'_lookup None using in_tmp by auto
    then show ?thesis
      unfolding filter_a2_map_def using tp'_ge m'_def ts_tp_lt'_new_tstp by auto
  next
    case (Some tstp')
    have Mapping.lookup m' xs = Some (max_tstp new_tstp tstp')
      unfolding m'_alt upd_set'_lookup Some using in_tmp by auto
    moreover have ts_tp_lt' ts' tp' (max_tstp new_tstp tstp')
  using max_tstp_intro''' ts_tp_lt'_new_tstp lookup_a2_map'[OF m_def Some] new_tstp_lt_isl
    by auto
    ultimately show ?thesis
      unfolding filter_a2_map_def using tp'_ge m'_def by auto
  qed
next
case (Some tp''')
then have tp'_gt: tp' > tp'''
  using xs_props(3)[unfolded proj_tuple_in_join_def] eqs(2)[unfolded filter_a1_map_def]
    False by (auto intro: Mapping_keys_intro)
define wtp where wtp ≡ max (tp - len) (tp''' + 1)
have wtp_xs_in: (wtp, xs) ∈ tmp
  unfolding wtp_def tmp_def using xs_props(1) Some False by fastforce
have wtp_le: wtp ≤ tp'
  using tp'_ge tp'_gt unfolding wtp_def by auto
have wtp_in: wtp ∈ {tp - len..tp}
  using tp'_lt_tp tp'_gt unfolding wtp_def by auto
have wtp_neg: tp + 1 ≠ wtp
  using wtp_in by auto
obtain m where m_def: Mapping.lookup a2_map wtp = Some m
  using wtp_in a2_map_keys Mapping_keys_dest by fastforce
obtain m' where m'_def: Mapping.lookup a2_map' wtp = Some m'
  using wtp_in a2_map'_keys Mapping_keys_dest by fastforce
have m'_alt: m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (wtp, b) ∈ tmp}
  using m'_def[unfolded a2_map'_def Mapping.lookup_update_neq[OF wtp_neg]
    upd_nested_lookup m_def] by auto
show ?thesis
proof (cases Mapping.lookup m xs)
  case None
  have Mapping.lookup m' xs = Some new_tstp
    using wtp_xs_in unfolding m'_alt upd_set'_lookup None by auto
  then show ?thesis
    unfolding filter_a2_map_def using wtp_le m'_def ts_tp_lt'_new_tstp by auto
next
  case (Some tstp')
  have Mapping.lookup m' xs = Some (max_tstp new_tstp tstp')

```

```

    using wtp_xs_in unfolding m'_alt upd_set' lookup Some by auto
    moreover have ts_tp_lt' ts' tp' (max_tstp new_tstp tstp')
using max_tstp_intro''' ts_tp_lt'_new_tstp lookup_a2_map'[OF m_def Some] new_tstp_lt_isl
    by auto
    ultimately show ?thesis
    using lookup_a2_map'[OF m_def Some] wtp_le m'_def
    unfolding filter_a2_map_def by auto
qed
qed
qed
qed
moreover have nt - t < left I  $\implies$  filter_a2_map I ts' tp' a2_map' = a2
proof (rule set_eqI, rule iffI)
  fix xs
  assume out: nt - t < left I
  assume xs_in: xs  $\in$  filter_a2_map I ts' tp' a2_map'
  then obtain m' tp'' tstp where m'_def: tp''  $\leq$  tp' Mapping.lookup a2_map' tp'' = Some m'
    Mapping.lookup m' xs = Some tstp ts_tp_lt' ts' tp' tstp
  unfolding filter_a2_map_def by (fastforce split: option.splits)
  have new_tstp_False: tstp = new_tstp  $\implies$  False
  using m'_def t_nt out tp'_lt_tp unfolding eqs(1)
  by (auto simp add: ts_tp_lt'_def new_tstp_def)
  show xs  $\in$  a2
  using filter_sub_a2[OF m'_def new_tstp_False xs_in] .
next
  fix xs
  assume xs  $\in$  a2
  then show xs  $\in$  filter_a2_map I ts' tp' a2_map'
  using a2_sub_filter by auto
qed
ultimately show triple_eq_pair (case tri of (t, a1, a2)  $\implies$ 
(t, if pos then join a1 True rel1 else a1  $\cup$  rel1,
if left I  $\leq$  nt - t then a2  $\cup$  join rel2 pos a1 else a2))
pair ( $\lambda$ tp'. filter_a1_map pos tp' a1_map') ( $\lambda$ ts' tp'. filter_a2_map I ts' tp' a2_map')
using eqs unfolding tri_def pair_def by auto
qed
have filter_a1_map_rel1: filter_a1_map pos tp a1_map' = rel1
  unfolding filter_a1_map_def a1_map'_def using leD lookup_a1_map_keys
  by (force simp add: a1_map_lookup less_imp_le_nat Mapping.lookup_filter
    Mapping_lookup_upd_set keys_is_none_rep dest: Mapping_keys_filterD
    intro: Mapping_keys_intro split: option.splits)
have filter_a1_map_rel2: filter_a2_map I nt tp a2_map' =
(if left I = 0 then rel2 else empty_table)
proof (cases left I = 0)
  case True
  note left_I_zero = True
  have  $\bigwedge$ tp' m' xs tstp. tp'  $\leq$  tp  $\implies$  Mapping.lookup a2_map' tp' = Some m'  $\implies$ 
    Mapping.lookup m' xs = Some tstp  $\implies$  ts_tp_lt' nt tp tstp  $\implies$  xs  $\in$  rel2
  proof -
    fix tp' m' xs tstp
    assume lassms: tp'  $\leq$  tp Mapping.lookup a2_map' tp' = Some m'
      Mapping.lookup m' xs = Some tstp ts_tp_lt' nt tp tstp
    have tp'_neq: tp + 1  $\neq$  tp'
      using lassms(1) by auto
    have tp'_in: tp'  $\in$  {tp - len..tp}
      using lassms(1,2) a2_map'_keys tp'_neq by (auto intro!: Mapping_keys_intro)
    obtain m where m_def: Mapping.lookup a2_map tp' = Some m

```

```

    m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp', b) ∈ tmp}
  using lassms(2)[unfolded a2_map'_def Mapping.lookup_update_neq[OF tp'_neq]
    upd_nested_lookup] tp'_in a2_map_keys
  by (fastforce dest: Mapping_keys_dest intro: Mapping_keys_intro split: option.splits)
have xs ∈ {b. (tp', b) ∈ tmp}
proof (rule ccontr)
  assume xs ∉ {b. (tp', b) ∈ tmp}
  then have Some: Mapping.lookup m xs = Some tstp
    using lassms(3)[unfolded m_def(2) upd_set'_lookup] by auto
  show False
    using lookup_a2_map'[OF m_def(1) Some] lassms(4)
    by (auto simp add: tstp_lt_def ts_tp_lt'_def split: sum.splits)
qed
then show xs ∈ rel2
  unfolding tmp_def by (auto split: option.splits if_splits)
qed
moreover have  $\bigwedge xs. xs \in rel2 \implies \exists m' \text{tstp. Mapping.lookup a2\_map}' tp = \text{Some } m' \wedge$ 
  Mapping.lookup m' xs = Some tstp  $\wedge ts\_tp\_lt' \text{nt } tp \text{tstp}$ 
proof -
  fix xs
  assume lassms: xs ∈ rel2
  obtain m' where m'_def: Mapping.lookup a2_map' tp = Some m'
    using a2_map'_keys by (fastforce dest: Mapping_keys_dest)
  have tp_neq: tp + 1 ≠ tp
    by auto
  obtain m where m_def: Mapping.lookup a2_map tp = Some m
    m' = upd_set' m new_tstp (max_tstp new_tstp) {b. (tp, b) ∈ tmp}
  using m'_def a2_map_keys unfolding a2_map'_def Mapping.lookup_update_neq[OF tp_neq]
    upd_nested_lookup
  by (auto dest: Mapping_keys_dest split: option.splits if_splits)
    (metis Mapping_keys_dest atLeastAtMost_iff diff_le_self le_eq_less_or_eq
      option.simps(3))
  have xs_in_tmp: xs ∈ {b. (tp, b) ∈ tmp}
    using lassms left_I_zero unfolding tmp_def by auto
  show  $\exists m' \text{tstp. Mapping.lookup a2\_map}' tp = \text{Some } m' \wedge$ 
    Mapping.lookup m' xs = Some tstp  $\wedge ts\_tp\_lt' \text{nt } tp \text{tstp}$ 
  proof (cases Mapping.lookup m xs)
    case None
    moreover have Mapping.lookup m' xs = Some new_tstp
      using xs_in_tmp unfolding m_def(2) upd_set'_lookup None by auto
    moreover have ts_tp_lt' nt tp new_tstp
      using left_I_zero new_tstp_def by (auto simp add: ts_tp_lt'_def)
    ultimately show ?thesis
      using xs_in_tmp m_def
      unfolding a2_map'_def Mapping.lookup_update_neq[OF tp_neq] upd_nested_lookup by auto
  next
    case (Some tstp')
    moreover have Mapping.lookup m' xs = Some (max_tstp new_tstp tstp')
      using xs_in_tmp unfolding m_def(2) upd_set'_lookup Some by auto
    moreover have ts_tp_lt' nt tp (max_tstp new_tstp tstp')
      using max_tstpE[of new_tstp tstp'] lookup_a2_map'[OF m_def(1) Some] new_tstp_lt_isl
      left_I_zero
      by (auto simp add: sum.discI(1) new_tstp_def ts_tp_lt'_def tstp_lt_def split: sum.splits)
    ultimately show ?thesis
      using xs_in_tmp m_def
      unfolding a2_map'_def Mapping.lookup_update_neq[OF tp_neq] upd_nested_lookup by auto
  qed
qed

```

```

ultimately show ?thesis
  using True by (fastforce simp add: filter_a2_map_def split: option.splits)
next
case False
note left_I_pos = False
have  $\bigwedge tp' m xs tstp. tp' \leq tp \implies Mapping.lookup\ a2\_map'\ tp' = Some\ m \implies Mapping.lookup\ m\ xs = Some\ tstp \implies \neg(ts\_tp\_lt'\ nt\ tp\ tstp)$ 
proof -
  fix tp' m' xs tstp
  assume lassms:  $tp' \leq tp \implies Mapping.lookup\ a2\_map'\ tp' = Some\ m'$ 
  Mapping.lookup m' xs = Some tstp
  from lassms(1) have tp'_neq_Suc_tp:  $tp + 1 \neq tp'$ 
  by auto
  show  $\neg(ts\_tp\_lt'\ nt\ tp\ tstp)$ 
proof (cases Mapping.lookup a2_map tp')
  case None
  then have tp'_in_tmp:  $tp' \in fst\ 'tmp$  and
  m'_alt:  $m' = upd\_set'\ Mapping.empty\ new\_tstp\ (max\_tstp\ new\_tstp)\ \{b.\ (tp',\ b) \in tmp\}$ 
  using lassms(2) unfolding a2_map'_def Mapping.lookup_update_neq[OF tp'_neq_Suc_tp]
  upd_nested_lookup by (auto split: if_splits)
  then have tstp = new_tstp
  using lassms(3)[unfolded m'_alt upd_set'_lookup]
  by (auto simp add: Mapping.lookup_empty split: if_splits)
  then show ?thesis
  using False by (auto simp add: ts_tp_lt'_def new_tstp_def split: if_splits sum.splits)
next
case (Some m)
then have m'_alt:  $m' = upd\_set'\ m\ new\_tstp\ (max\_tstp\ new\_tstp)\ \{b.\ (tp',\ b) \in tmp\}$ 
  using lassms(2) unfolding a2_map'_def Mapping.lookup_update_neq[OF tp'_neq_Suc_tp]
  upd_nested_lookup by auto
note lookup_a2_map_tp' = Some
show ?thesis
proof (cases Mapping.lookup m xs)
  case None
  then have tstp = new_tstp
  using lassms(3) unfolding m'_alt upd_set'_lookup by (auto split: if_splits)
  then show ?thesis
  using False by (auto simp add: ts_tp_lt'_def new_tstp_def split: if_splits sum.splits)
next
case (Some tstp')
show ?thesis
proof (cases  $xs \in \{b.\ (tp',\ b) \in tmp\}$ )
  case True
  then have tstp_eq:  $tstp = max\_tstp\ new\_tstp\ tstp'$ 
  using lassms(3)
  unfolding m'_alt upd_set'_lookup Some by auto
  show ?thesis
  using max_tstpE[of new_tstp tstp'] lookup_a2_map'[OF lookup_a2_map_tp' Some]
  new_tstp_lt_isl left_I_pos
  by (auto simp add: tstp_eq tstp_lt_def ts_tp_lt'_def split: sum.splits)
next
case False
then show ?thesis
using lassms(3) lookup_a2_map'[OF lookup_a2_map_tp' Some]
unfolding m'_alt upd_set'_lookup Some
by (auto simp add: ts_tp_lt'_def tstp_lt_def split: sum.splits)
qed
qed

```

```

    qed
  qed
  then show ?thesis
    using False by (auto simp add: filter_a2_map_def empty_table_def split: option.splits)
  qed
  have zip_dist: zip (linearize tss @ [nt]) ([tp - len..<tp] @ [tp]) =
    zip (linearize tss) [tp - len..<tp] @ [(nt, tp)]
    using valid_shift_aux(1) by auto
  have list_all2': list_all2 ( $\lambda x y. triple\_eq\_pair\ x\ y\ (\lambda tp'. filter\_a1\_map\ pos\ tp'\ a1\_map')$ )
    ( $\lambda ts' tp'. filter\_a2\_map\ I\ ts'\ tp'\ a2\_map'$ )
    (drop (length done) (update_until args rel1 rel2 nt auxlist))
    (zip (linearize tss') [tp + 1 - (len + 1)..<tp + 1])
    unfolding lin_tss' tp_upt_Suc drop_update_until zip_dist
    using filter_a1_map_rel1 filter_a1_map_rel2 list_all2_appendI[OF list_all2_old]
    by auto
  show ?thesis
    using valid_shift_aux len_lin_tss' sorted_lin_tss' set_lin_tss' tab_a1_map'_keys a2_map'_keys'
      len_upd_until sorted_upd_until lookup_a1_map_keys' rev_done' set_take_auxlist'
      lookup_a2_map'_keys list_all2'
    unfolding valid_mmuaux_def add_new_mmuaux_eq valid_mmuaux'.simps
      I_def n_def L_def R_def pos_def by auto
  qed

lemma list_all2_check_before: list_all2 ( $\lambda x y. triple\_eq\_pair\ x\ y\ f\ g$ ) xs (zip ys zs)  $\implies$ 
  ( $\bigwedge y. y \in set\ ys \implies \neg enat\ y + right\ I < nt$ )  $\implies x \in set\ xs \implies \neg check\_before\ I\ nt\ x$ 
  by (auto simp: in_set_zip elim!: list_all2_in_setE triple_eq_pair.elims)

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  $\geq$  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 -
  define I where I = args_ivl args
  define pos where pos = args_pos args
  have valid_folded: valid_mmuaux' args cur nt aux auxlist
    using assms(1,2) valid_mmuaux'_mono unfolding valid_mmuaux_def by blast
  obtain tp len tss maskL maskR a1_map a2_map done done_length where shift_aux_def:
    shift_mmuaux args nt aux = (tp, tss, len, maskL, maskR, a1_map, a2_map, done, done_length)
    by (cases shift_mmuaux args nt aux) auto
  have valid_shift_aux: valid_mmuaux' args cur nt (tp, tss, len, maskL, maskR,
    a1_map, a2_map, done, done_length) auxlist
     $\bigwedge ts. ts \in set\ (linearize\ tss) \implies \neg enat\ ts + right\ (args\_ivl\ args) < enat\ nt$ 
    using valid_shift_mmuaux'[OF assms(1)[unfolded valid_mmuaux_def] assms(2)]
    unfolding shift_aux_def by auto
  have len_done_auxlist: length done  $\leq$  length auxlist
    using valid_shift_aux by auto
  have list_all:  $\bigwedge x. x \in set\ (take\ (length\ done)\ auxlist) \implies check\_before\ I\ nt\ x$ 
    using valid_shift_aux unfolding I_def by auto
  have set_drop_auxlist:  $\bigwedge x. x \in set\ (drop\ (length\ done)\ auxlist) \implies \neg check\_before\ I\ nt\ x$ 
    using valid_shift_aux[unfolded valid_mmuaux'.simps]
    list_all2_check_before[OF valid_shift_aux(2)] unfolding I_def by fast
  have take_auxlist_takeWhile: take (length done) auxlist = takeWhile (check_before I nt) auxlist

```

```

    using len_done_auxlist list_all set_drop_auxlist
    by (rule take_takeWhile) assumption+
  have rev_done: rev done = map proj_thd (take (length done) auxlist)
    using valid_shift_aux by auto
  then have res'_def: res' = rev done
    using eval_until_res[OF assms(4)] unfolding take_auxlist_takeWhile I_def by auto
  then have auxlist'_def: auxlist' = drop (length done) auxlist
    using eval_until_auxlist'[OF assms(4)] by auto
  have eval_mmuaux_eq: eval_mmuaux args nt aux = (rev done, (tp, tss, len, maskL, maskR,
    a1_map, a2_map, [], 0))
    using shift_aux_def by auto
  show ?thesis
    using assms(3) done_empty_valid_mmuaux'_intro[OF valid_shift_aux(1)]
    unfolding shift_aux_def eval_mmuaux_eq pos_def auxlist'_def res'_def valid_mmuaux_def by auto
qed

```

```

definition init_mmuaux :: args ⇒ '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:  $L \subseteq R \implies$  valid_mmuaux (init_args I n L R b) 0
  (init_mmuaux (init_args I n L R b)) []
  unfolding init_mmuaux_def valid_mmuaux_def
  by (auto simp add: init_args_def empty_queue_rep table_def Mapping.lookup_update)

```

```

fun length_mmuaux :: args ⇒ 'a mmuaux ⇒ 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
  using assms by (cases aux) (auto simp add: valid_mmuaux_def dest: list_all2_lengthD)

```

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 ..
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 *mmuau* **where**
add_new_mmuau' = *add_new_mmuau*

interpretation *muau* *valid_mmuau* *init_mmuau* *add_new_mmuau*' *length_mmuau* *eval_mmuau*
using *valid_init_mmuau* *valid_add_new_mmuau* *valid_length_mmuau* *valid_eval_mmuau*
unfolding *add_new_mmuau*'_def
by *unfold_locales* *assumption*+

type_synonym 'a *vmsau* = nat × (nat × 'a table) list

definition *valid_vmsau* :: args ⇒ nat ⇒ *event_data* *vmsau* ⇒
(nat × *event_data* table) list ⇒ bool **where**
valid_vmsau = (λ_ cur (t, au) auclist. t = cur ∧ au = auclist)

definition *init_vmsau* :: args ⇒ *event_data* *vmsau* **where**
init_vmsau = (λ_. (0, []))

definition *add_new_ts_vmsau* :: args ⇒ nat ⇒ *event_data* *vmsau* ⇒ *event_data* *vmsau* **where**
add_new_ts_vmsau = (λargs nt (t, auclist). (nt, filter (λ(t, rel).
enat (nt - t) ≤ right (args_ivl args)) auclist))

definition *join_vmsau* :: args ⇒ *event_data* table ⇒ *event_data* *vmsau* ⇒ *event_data* *vmsau* **where**
join_vmsau = (λargs rel1 (t, auclist). (t, map (λ(t, rel).
(t, join_rel (args_pos args) rel1)) auclist))

definition *add_new_table_vmsau* :: args ⇒ *event_data* table ⇒ *event_data* *vmsau* ⇒
event_data *vmsau* **where**
add_new_table_vmsau = (λargs rel2 (cur, auclist). (cur, (case auclist of
[] => [(cur, rel2)]
| ((t, y) # ts) => if t = cur then (t, y ∪ rel2) # ts else (cur, rel2) # auclist)))

definition *result_vmsau* :: args ⇒ *event_data* *vmsau* ⇒ *event_data* table **where**
result_vmsau = (λargs (cur, auclist).
foldr (∪) [rel. (t, rel) ← auclist, left (args_ivl args) ≤ cur - t] {})

type_synonym 'a *vmuau* = nat × (nat × 'a table × 'a table) list

definition *valid_vmuau* :: args ⇒ nat ⇒ *event_data* *vmuau* ⇒
(nat × *event_data* table × *event_data* table) list ⇒ bool **where**
valid_vmuau = (λ_ cur (t, au) auclist. t = cur ∧ au = auclist)

definition *init_vmuau* :: args ⇒ *event_data* *vmuau* **where**
init_vmuau = (λ_. (0, []))

definition *add_new_vmuau* :: args ⇒ *event_data* table ⇒ *event_data* table ⇒ nat ⇒
event_data *vmuau* ⇒ *event_data* *vmuau* **where**
add_new_vmuau = (λargs rel1 rel2 nt (t, auclist). (nt, update_until args rel1 rel2 nt auclist))

definition *length_vmuau* :: args ⇒ *event_data* *vmuau* ⇒ nat **where**
length_vmuau = (λ_ (_, auclist). length auclist)

definition *eval_vmuau* :: args ⇒ nat ⇒ *event_data* *vmuau* ⇒
event_data table list × *event_data* *vmuau* **where**
eval_vmuau = (λargs nt (t, auclist).
(let (res, auclist') = eval_until (args_ivl args) nt auclist in (res, (t, auclist'))))

global_interpretation *verimon_mau*: *mau* *valid_vmsau* *init_vmsau* *add_new_ts_vmsau* *join_vmsau*
add_new_table_vmsau *result_vmsau* *valid_vmuau* *init_vmuau* *add_new_vmuau* *length_vmuau*

```

eval_vmuaux
defines vminit0 = maux.minit0 (init_vmsaux :: _ ⇒ event_data vmsaux) (init_vmuaux :: _ ⇒
event_data vmuaux) :: _ ⇒ Formula.formula ⇒ _
and vminit = maux.minit (init_vmsaux :: _ ⇒ event_data vmsaux) (init_vmuaux :: _ ⇒ event_data
vmuaux) :: Formula.formula ⇒ _
and vminit_safe = maux.minit_safe (init_vmsaux :: _ ⇒ event_data vmsaux) (init_vmuaux :: _ ⇒
event_data vmuaux) :: Formula.formula ⇒ _
and vmupdate_since = maux.update_since add_new_ts_vmsaux join_vmsaux add_new_table_vmsaux
(result_vmsaux :: _ ⇒ event_data vmsaux ⇒ event_data table)
and vmeval = maux.meval add_new_ts_vmsaux join_vmsaux add_new_table_vmsaux (result_vmsaux
:: _ ⇒ event_data vmsaux ⇒ _) add_new_vmuaux (eval_vmuaux :: _ ⇒ _ ⇒ event_data vmuaux ⇒
_)
and vmstep = maux.mstep add_new_ts_vmsaux join_vmsaux add_new_table_vmsaux (result_vmsaux
:: _ ⇒ event_data vmsaux ⇒ _) add_new_vmuaux (eval_vmuaux :: _ ⇒ _ ⇒ event_data vmuaux ⇒
_)
and vmsteps0_stateless = maux.msteps0_stateless add_new_ts_vmsaux join_vmsaux add_new_table_vmsaux
(result_vmsaux :: _ ⇒ event_data vmsaux ⇒ _) add_new_vmuaux (eval_vmuaux :: _ ⇒ _ ⇒ event_data
vmuaux ⇒ _)
and vmsteps_stateless = maux.msteps_stateless add_new_ts_vmsaux join_vmsaux add_new_table_vmsaux
(result_vmsaux :: _ ⇒ event_data vmsaux ⇒ _) add_new_vmuaux (eval_vmuaux :: _ ⇒ _ ⇒ event_data
vmuaux ⇒ _)
and vmonitor = maux.monitor init_vmsaux add_new_ts_vmsaux join_vmsaux add_new_table_vmsaux
(result_vmsaux :: _ ⇒ event_data vmsaux ⇒ _) init_vmuaux add_new_vmuaux (eval_vmuaux :: _ ⇒
_ ⇒ event_data vmuaux ⇒ _)
unfolding valid_vmsaux_def init_vmsaux_def add_new_ts_vmsaux_def join_vmsaux_def
add_new_table_vmsaux_def result_vmsaux_def valid_vmuaux_def init_vmuaux_def add_new_vmuaux_def
length_vmuaux_def eval_vmuaux_def
by unfold_locales auto

global_interpretation default_maux: maux valid_mmsaux init_mmsaux :: _ ⇒ event_data mmsaux
add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux result_mmsaux
valid_mmuaux init_mmuaux :: _ ⇒ event_data mmuaux add_new_mmuaux' length_mmuaux eval_mmuaux
defines minit0 = maux.minit0 (init_mmsaux :: _ ⇒ event_data mmsaux) (init_mmuaux :: _ ⇒
event_data mmuaux) :: _ ⇒ Formula.formula ⇒ _
and minit = maux.minit (init_mmsaux :: _ ⇒ event_data mmsaux) (init_mmuaux :: _ ⇒ event_data
mmuaux) :: Formula.formula ⇒ _
and minit_safe = maux.minit_safe (init_mmsaux :: _ ⇒ event_data mmsaux) (init_mmuaux :: _ ⇒
event_data mmuaux) :: Formula.formula ⇒ _
and mupdate_since = maux.update_since add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux
(result_mmsaux :: _ ⇒ event_data mmsaux ⇒ event_data table)
and meval = maux.meval add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux (result_mmsaux
:: _ ⇒ event_data mmsaux ⇒ _) add_new_mmuaux' (eval_mmuaux :: _ ⇒ _ ⇒ event_data mmuaux
⇒ _)
and mstep = maux.mstep add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux (result_mmsaux
:: _ ⇒ event_data mmsaux ⇒ _) add_new_mmuaux' (eval_mmuaux :: _ ⇒ _ ⇒ event_data mmuaux
⇒ _)
and msteps0_stateless = maux.msteps0_stateless add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux
(result_mmsaux :: _ ⇒ event_data mmsaux ⇒ _) add_new_mmuaux' (eval_mmuaux :: _ ⇒ _ ⇒
event_data mmuaux ⇒ _)
and msteps_stateless = maux.msteps_stateless add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux
(result_mmsaux :: _ ⇒ event_data mmsaux ⇒ _) add_new_mmuaux' (eval_mmuaux :: _ ⇒ _ ⇒
event_data mmuaux ⇒ _)
and monitor = maux.monitor init_mmsaux add_new_ts_mmsaux gc_join_mmsaux add_new_table_mmsaux
(result_mmsaux :: _ ⇒ event_data mmsaux ⇒ _) init_mmuaux add_new_mmuaux' (eval_mmuaux ::
_ ⇒ _ ⇒ event_data mmuaux ⇒ _)
by unfold_locales

```

lemma image_these: $f \text{ ' } \text{Option.these } X = \text{Option.these } (\text{map_option } f \text{ ' } X)$

by (force simp: in_these_eq Bex_def image_iff map_option_case split: option.splits)

thm *default_maux.meval.simps(2)*

lemma *meval_MPred*: *meval n t db (MPred e ts) =*
 (case Mapping.lookup db e of None \Rightarrow $\{\}$ | Some Xs \Rightarrow map $(\lambda X. \bigcup v \in X.$
 (set_option (map_option $(\lambda f. \text{Table.tabulate } f \ 0 \ n)$ (match ts v)))) Xs, MPred e ts)
 by (force split: option.splits simp: Option.these_def image_iff)

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 **where**
mk_db t = Monitor.mk_db $(\bigcup n \in \text{set } (map \text{fst } t). (\lambda v. (n, v)) \text{ 'the } (map_of \ t \ n))$

definition *rbt_fold* :: $_ \Rightarrow$ event_data tuple set_rbt \Rightarrow $_ \Rightarrow$ $_ \text{ where}$
rbt_fold = RBT_Set2.fold

definition *rbt_empty* :: event_data list set_rbt **where**
rbt_empty = RBT_Set2.empty

definition *rbt_insert* :: $_ \Rightarrow$ $_ \Rightarrow$ event_data list set_rbt **where**
rbt_insert = RBT_Set2.insert

lemma *saturate_commute*:
assumes $\bigwedge s. r \in g \ s \ \bigwedge s. g \ (\text{insert } r \ s) = g \ s \ \bigwedge s. r \in s \implies h \ s = g \ s$
and *terminates*: $\text{mono } g \ \bigwedge X. X \subseteq C \implies g \ X \subseteq C \ \text{finite } C$
shows *saturate g* $\{\}$ = *saturate h* $\{r\}$
proof (cases *g* $\{\}$ = $\{r\}$)
case True
with *assms* **have** *g* $\{r\}$ = $\{r\}$ *h* $\{r\}$ = $\{r\}$ **by** *auto*
with True **show** ?thesis
by (subst (1 2) *saturate_code*; subst *saturate_code*) (*simp add: Let_def*)
next
case False
then show ?thesis
unfolding *saturate_def* *while_def*
using *while_option_finite_subset_Some[OF terminates]* *assms(1-3)*
by (subst *while_option_commute_invariant*[of $\lambda S. S = \{\}$ $\vee r \in S \ \lambda S. g \ S \neq S \ g \ \lambda S. h \ S \neq S \ \text{insert } r \ h \ \{\}$, *symmetric*])
 (*auto 4 4 dest: while_option_stop*[of $\lambda S. g \ S \neq S \ g \ \{\}$])
qed

definition *RPDs_aux* = *saturate* $(\lambda S. S \cup \bigcup (RPD \text{ ' } S))$

lemma *RPDs_aux_code[code]*:
RPDs_aux S = (let *S'* = $S \cup \text{Set.bind } S \ \text{RPD}$ in if $S' \subseteq S$ then *S* else *RPDs_aux S'*)
unfolding *RPDs_aux_def* *bind_UNION*
by (subst *saturate_code*) *auto*

declare *RPDs_code[code del]*

lemma *RPDs_code[code]*: *RPDs r* = *RPDs_aux* $\{r\}$
unfolding *RPDs_aux_def* *RPDs_code*
by (rule *saturate_commute*[**where** $C = \text{RPDs } r$])
 (*auto 0 3 simp: mono_def subset_singleton_iff RPDs_refl RPDs_trans finite_RPDs*)

definition *LPDs_aux* = *saturate* $(\lambda S. S \cup \bigcup (LPD \text{ ' } S))$

lemma *LPDs_aux_code[code]*:

```

LPDs_aux S = (let S' = S ∪ Set.bind S LPD in if S' ⊆ S then S else LPDs_aux S')
unfolding LPDs_aux_def bind_UNION
by (subst saturate_code) auto

declare LPDs_code[code del]
lemma LPDs_code[code]: LPDs r = LPDs_aux {r}
unfolding LPDs_aux_def LPDs_code
by (rule saturate_commute[where C=LPDs r])
    (auto 0 3 simp: mono_def subset_singleton_iff LPDs_refl LPDs_trans finite_LPDS)

lemma is_empty_table_unfold [code_unfold]:
X = empty_table ↔ Set.is_empty X
empty_table = X ↔ Set.is_empty X
set_eq X empty_table ↔ Set.is_empty X
set_eq empty_table X ↔ Set.is_empty X
X = (set_empty impl) ↔ Set.is_empty X
(set_empty impl) = X ↔ Set.is_empty X
set_eq X (set_empty impl) ↔ Set.is_empty X
set_eq (set_empty impl) X ↔ Set.is_empty X
unfolding set_eq_def set_empty_def empty_table_def by auto

lemma tabulate_rbt_code[code]: Monitor.mrtabulate (xs :: mregex list) f =
(case ID CCOMPARE(mregex) of None ⇒ Code.abort (STR "tabulate RBT_Mapping: ccompare =
None") (λ_. Monitor.mrtabulate (xs :: mregex list) f)
| _ ⇒ RBT_Mapping (RBT_Mapping2.bulkload (List.map_filter (λk. let fk = f k in if fk = empty_table
then None else Some (k, fk)) xs)))
unfolding mrtabulate.abs_eq RBT_Mapping_def
by (auto split: option.splits)

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 ⇒ Code.abort (STR "combine RBT_Mapping: ccompare = None")
(λ_. Mapping.combine f (RBT_Mapping t) (RBT_Mapping u))
| Some _ ⇒ RBT_Mapping (RBT_Mapping2.join (λ_. f) t u))
by (auto simp add: Mapping.combine.abs_eq Mapping_inject lookup_join split: option.split)

lemma upd_set_empty[simp]: upd_set m f {} = m
by transfer auto

lemma upd_set_insert[simp]: upd_set m f (insert x A) = Mapping.update x (f x) (upd_set m f A)
by (rule mapping_eqI) (auto simp: Mapping_lookup_upd_set Mapping_lookup_update')

lemma upd_set_fold:
assumes finite A
shows upd_set m f A = Finite_Set.fold (λa. Mapping.update a (f a)) m A
proof –
interpret comp_fun_idem λa. Mapping.update a (f a)
by unfold_locales (transfer; auto simp: fun_eq_iff)+
from assms show ?thesis
by (induct A arbitrary: m rule: finite.induct) auto
qed

lift_definition upd_cfi :: ('a ⇒ 'b) ⇒ ('a, ('a, 'b) mapping) comp_fun_idem
is λf a m. Mapping.update a (f a) m
by unfold_locales (transfer; auto simp: fun_eq_iff)+

lemma upd_set_code[code]:

```


definition *upd_nested_max_tstp* **where**
upd_nested_max_tstp *m d X* = *upd_nested* *m d* (*max_tstp* *d*) *X*

lemma *upd_nested_max_tstp_fold*:
assumes *finite X*
shows *upd_nested_max_tstp* *m d X* = *Finite_Set.fold* (*upd_nested_step* *d* (*max_tstp* *d*)) *m X*

proof –
interpret *comp_fun_idem upd_nested_step* *d* (*max_tstp* *d*)
by (*unfold_locales*; *rule ext*)
(auto simp add: comp_def upd_nested_step_def Mapping.lookup_update' Mapping.lookup_empty
update_update max_tstp_d_d max_tstp_idem' split: option.splits)
note *upd_nested_insert'* = *upd_nested_insert*[*of d max_tstp d*,
OF max_tstp_d_d[symmetric] max_tstp_idem']
show *?thesis*
using *assms*
by (*induct X arbitrary: m rule: finite.induct*)
(auto simp add: upd_nested_max_tstp_def upd_nested_insert')

qed

lift_definition *upd_nested_max_tstp_cfi* ::
ts + tp \Rightarrow (*'a* \times *'b*, (*'a*, (*'b*, *ts + tp*) *mapping*) *comp_fun_idem*
is $\lambda d.$ *upd_nested_step* *d* (*max_tstp* *d*)
by (*unfold_locales*; *rule ext*)
(auto simp add: comp_def upd_nested_step_def Mapping.lookup_update' Mapping.lookup_empty
update_update max_tstp_d_d max_tstp_idem' split: option.splits)

lemma *upd_nested_max_tstp_code*[*code*]:
upd_nested_max_tstp *m d X* = (*if finite X then set_fold_cfi* (*upd_nested_max_tstp_cfi* *d*) *m X*
else Code.abort (*STR "upd_nested_max_tstp: infinite"*) ($\lambda_.$ *upd_nested_max_tstp* *m d X*))
by *transfer* (*auto simp add: upd_nested_max_tstp_fold*)

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*
unfolding *filter_set_def*
by *transfer* (*auto simp: fun_eq_iff split: option.splits*)

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* \Rightarrow *if* *t* = *u* *then Mapping.delete* *x m'* *else m' | _* \Rightarrow *m'*)
unfolding *filter_set_def*
by *transfer* (*auto simp: fun_eq_iff Let_def Map_To_Mapping.map_apply_def split: option.splits*)

lemma *filter_set_fold*:
assumes *finite A*
shows *filter_set* *m A t* = *Finite_Set.fold* ($\lambda a m.$
case Mapping.lookup *m a* *of Some u* \Rightarrow *if* *t* = *u* *then Mapping.delete* *a m* *else m | _* \Rightarrow *m*) *m A*

proof –
interpret *comp_fun_idem* $\lambda a m.$
case Mapping.lookup *m a* *of Some u* \Rightarrow *if* *t* = *u* *then Mapping.delete* *a m* *else m | _* \Rightarrow *m*
by *unfold_locales*
(transfer; auto simp: fun_eq_iff Map_To_Mapping.map_apply_def split: option.splits)
from *assms* **show** *?thesis*
by (*induct A arbitrary: m rule: finite.induct*) (*auto simp: Let_def*)

qed

lift_definition *filter_cfi* :: *'b* \Rightarrow (*'a*, (*'a*, *'b*) *mapping*) *comp_fun_idem*

is $\lambda t a m$.
 case *Mapping.lookup* *m a* of *Some u* \Rightarrow if $t = u$ then *Mapping.delete a m* else $m \mid _ \Rightarrow m$
 by *unfold_locales*
 (transfer; auto simp: *fun_eq_iff Map_To_Mapping.map_apply_def split: option.splits*)+

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")*) ($\lambda _.$ *filter_set m A t*)
 by (transfer fixing: *m*) (auto simp: *filter_set_fold*)

lemma *filter_Mapping*[code]:
fixes $t :: ('a :: ccompare, 'b)$ *mapping_rbt* **shows**
Mapping.filter P (RBT_Mapping t) =
 (case *ID CCOMPARE('a)* of *None* \Rightarrow *Code.abort (STR "filter RBT_Mapping: ccompare = None")*)
 ($\lambda _.$ *Mapping.filter P (RBT_Mapping t)*)
 | *Some _* \Rightarrow *RBT_Mapping (RBT_Mapping2.filter (case_prod P) t)*)
 by (auto simp add: *Mapping.filter.abs_eq Mapping_inject split: option.split*)

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*
unfolding *filter_join_def*
 by transfer (auto split: *option.splits*)

lemma *filter_join_False_insert*: *filter_join False (insert a A) m* =
filter_join False A (Mapping.delete a m)

proof –
 {
fix x
have *Mapping.lookup (filter_join False (insert a A) m) x* =
Mapping.lookup (filter_join False A (Mapping.delete a m)) x
by (auto simp add: *filter_join_def Mapping.lookup_filter Mapping_lookup_delete split: option.splits*)
 }
then show ?thesis
 by (simp add: *mapping_eqI*)
qed

lemma *filter_join_False*:
assumes *finite A*
shows *filter_join False A m* = *Finite_Set.fold Mapping.delete m A*
proof –
interpret *comp_fun_idem Mapping.delete*
by (*unfold_locales; transfer*) (*fastforce simp add: comp_def*)
from *assms* **show** ?thesis
by (*induction A arbitrary: m rule: finite.induct*)
 (*auto simp add: filter_join_False_empty filter_join_False_insert fold_fun_left_comm*)
qed

lift_definition *filter_not_in_cfi* :: $('a, ('a, 'b)$ *mapping)* *comp_fun_idem* **is** *Mapping.delete*
by (*unfold_locales; transfer*) (*fastforce simp add: comp_def*)

lemma *filter_join_code*[code]:
filter_join pos A m =
 (if $\neg pos \wedge finite A \wedge card A < Mapping.size m$ then *set_fold_cfi filter_not_in_cfi m A*
 else *Mapping.filter (join_filter_cond pos A) m*)

unfolding *filter_join_def*
by (transfer fixing: m) (use filter_join_False in ⟨auto simp add: filter_join_def⟩)

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 λb a. a - {b}
by *unfold_locales auto*

lemma *set_minus_finite*:
assumes *fin*: finite Y
shows *set_minus* X Y = *Finite_Set.fold* (λa X. X - {a}) X Y
proof -
interpret *comp_fun_idem* λa X. X - {a}
by *unfold_locales auto*
from *assms show ?thesis*
by (*induction* Y *arbitrary*: X *rule*: *finite.induct*) (*auto simp add*: *set_minus_def*)
qed

lemma *set_minus_code*[*code*]: *set_minus* X Y =
(*if* finite Y ∧ *card* Y < *card* X *then set_fold_cfi remove_cfi* X Y *else* X - Y)
by transfer (use *set_minus_finite* in ⟨*auto simp add*: *set_minus_def*⟩)

declare *bin_join.simps*[*folded set_minus_def, code*]

definition *remove_Union* **where**
remove_Union A X B = A - (⋃ x ∈ X. B x)

lemma *remove_Union_finite*:
assumes finite X
shows *remove_Union* A X B = *Finite_Set.fold* (λx A. A - B x) A X
proof -
interpret *comp_fun_idem* λx A. A - B x
by *unfold_locales auto*
from *assms show ?thesis*
by (*induct* X *arbitrary*: A *rule*: *finite_induct*) (*auto simp*: *remove_Union_def*)
qed

lift_definition *remove_Union_cfi* :: ('a ⇒ 'b set) ⇒ ('a, 'b set) comp_fun_idem **is** λB x A. A - B x
by *unfold_locales auto*

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 - (⋃ x ∈ X. B x))
by (transfer fixing: A X B) (use *remove_Union_finite*[*of* X A B] in ⟨*auto simp add*: *remove_Union_def*⟩)

lemma *tabulate_remdups*: *Mapping.tabulate* xs f = *Mapping.tabulate* (*remdups* xs) f
by (transfer fixing: xs f) (*auto simp*: *map_of_map_restrict*)

lift_definition *clearjunk* :: (String.literal × event_data list set) list ⇒ (String.literal, event_data list set list) alist **is**
λt. *List.map_filter* (λ(p, X). *if* X = {} *then* None *else* Some (p, [X])) (*AList.clearjunk* t)
unfolding *map_filter_def o_def list.map_comp*
by (*subst map_cong*[*OF refl, of _ _ fst*]) (*auto simp*: *map_filter_def distinct_map fst_filter split*:
if_splits)

lemma *map_filter_snd_map_filter*: *List.map_filter* (λ(a, b). *if* P b *then* None *else* Some (f a b)) xs =
map (λ(a, b). f a b) (*filter* (λx. ¬ P (snd x)) xs)

by (*simp add: map_filter_def prod.case_eq_if*)

lemma *mk_db_code_alist*:

mk_db t = Assoc_List_Mapping (clearjunk t)

unfolding *mk_db_def Assoc_List_Mapping_def*

by (*transfer' fixing: t*)

(*auto simp: map_filter_snd_map_filter fun_eq_iff map_of_map image_iff map_of_clearjunk*

map_of_filter_apply dest: weak_map_of_SomeI intro!: beXI[rotated, OF map_of_SomeD]

split: if_splits option.splits)

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*))

unfolding *mk_db_def*

by (*transfer' fixing: t*) (*auto simp: map_filter_snd_map_filter fun_eq_iff map_of_map image_iff*

map_of_clearjunk map_of_filter_apply dest: weak_map_of_SomeI intro!: beXI[rotated, OF map_of_SomeD]

split: if_splits option.splits)

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.