

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

Thibault Dardinier    Lukas Heimes    Martin Raszyk    Joshua Schneider  
Dmitriy Traytel

June 14, 2026

## Abstract

A monitor is a runtime verification tool that solves the following problem: Given a stream of time-stamped events and a policy formulated in a specification language, decide whether the policy is satisfied at every point in the stream. We verify the correctness of an executable monitor for specifications given as formulas in metric first-order dynamic logic (MFODL), which combines the features of metric first-order temporal logic (MFOTL) [2] and metric dynamic logic [3]. Thus, MFODL supports real-time constraints, first-order parameters, and regular expressions. Additionally, the monitor supports aggregation operations such as count and sum. This formalization, which is described in a paper at IJCAR 2020 [1], significantly extends [previous work on a verified monitor](#) for MFOTL [4]. Apart from the addition of regular expressions and aggregations, we implemented [multi-way joins](#) and a specialized sliding window algorithm to further optimize the monitor.

## Contents

<b>1</b>	<b>Code adaptation for IEEE double-precision floats</b>	<b>2</b>
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
<b>2</b>	<b>Event parameters</b>	<b>11</b>
<b>3</b>	<b>Regular expressions</b>	<b>12</b>
<b>4</b>	<b>Metric first-order dynamic logic</b>	<b>19</b>
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
<b>5</b>	<b>Optimized relational join</b>	<b>41</b>
5.1	Binary join . . . . .	41
5.2	Multi-way join . . . . .	43

<b>6</b>	<b>Generic monitoring algorithm</b>	<b>52</b>
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
<b>7</b>	<b>Efficient implementation of temporal operators</b>	<b>152</b>
7.1	Optimized queue data structure . . . . .	152
7.2	Optimized data structure for Since . . . . .	156
7.3	Optimized data structure for Until . . . . .	178
<b>8</b>	<b>Instantiation of the generic algorithm and code setup</b>	<b>205</b>

## 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\_SucE[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**  
 $\lambda x. \text{zerosign } 0$  (*intround RNE (int\_of\_integer x)*) .

**definition** *double\_of\_int* **where** *double\_of\_int x* = *double\_of\_integer (integer\_of\_int x)*

**lemma** [*code*]: *double\_of\_int (int\_of\_integer x)* = *double\_of\_integer x*  
**unfolding** *double\_of\_int\_def* **by** *simp*

**lift\_definition** *integer\_of\_double* :: *double*  $\Rightarrow$  *integer* **is**  
 $\lambda x. \text{if } \text{IEEE.is\_nan } x \vee \text{IEEE.is\_infinity } x \text{ then undefined}$   
 $\text{else integer\_of\_int } \lfloor \text{valof (intround roundTowardZero (valof } x \text{))} \rfloor$  (*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*:  $\neg \text{is\_nan } x \implies \text{fcompare\_double } x x = \text{ccode.Eq}$   
**by** *transfer (auto simp: fcompare\_def)*

**lemma** *fcompare\_double\_Eq1*: *fcompare\_double x y* = *ccode.Eq*  $\implies \text{fcompare\_double } y z = c \implies \text{fcompare\_double } x z = c$   
**by** *transfer (auto simp: fcompare\_def split: if\_splits)*

**lemma** *fcompare\_double\_Eq2*: *fcompare\_double y z* = *ccode.Eq*  $\implies \text{fcompare\_double } x y = c \implies \text{fcompare\_double } x z = c$   
**by** *transfer (auto simp: fcompare\_def split: if\_splits)*

**lemma** *fcompare\_double\_Lt\_trans*: *fcompare\_double x y* = *ccode.Lt*  $\implies \text{fcompare\_double } y z = \text{ccode.Lt} \implies \text{fcompare\_double } x z = \text{ccode.Lt}$   
**by** *transfer (auto simp: fcompare\_def split: if\_splits)*

**lemma** *fcompare\_double\_eq*:  $\neg \text{is\_zero } x \vee \neg \text{is\_zero } y \implies \text{fcompare\_double } x y = \text{ccode.Eq} \implies x = y$   
**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*  $\implies \text{fcompare\_double } y x = \text{ccode.Lt} \implies \text{False}$   
**by** *transfer (auto simp: fcompare\_def split: if\_splits)*

**lemma** *compare\_double\_swap*:  $0 < \text{compare\_double } x y \longleftrightarrow \text{compare\_double } y x < 0$   
**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*)<sup>+</sup>

**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*  $\varphi$ ) = *fv*  $\varphi$   
| *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*  $\varphi$ ) = *Max* (*insert* 0 (*Suc* ' *fv*  $\varphi$ ))  
*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*  $\varphi$ ) =  $(\lambda i j. i = j \wedge \text{test } i \varphi)$   
| *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*]:

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

by (induct r arbitrary: r') auto

**qualified primrec eps where**

eps test i (Skip n) = (n = 0)  
| eps test i (Test  $\varphi$ ) = test i  $\varphi$   
| 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  $\varphi$ ) = {}  
| 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 $\kappa$  where**

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

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

**lemma lpd $\kappa$ \_lpd:** lpd $\kappa$   $\kappa$  test i r =  $\kappa$  ' lpd test i r

by (induct r arbitrary:  $\kappa$ ) (auto simp: TimesR\_def split: nat.splits)

**lemma rpd $\kappa$ \_rpd:** rpd $\kappa$   $\kappa$  test i r =  $\kappa$  ' rpd test i r

by (induct r arbitrary:  $\kappa$ ) (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  $\text{match test } (Times \ r \ s) \ i \ j \longleftrightarrow (\exists k. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j)$   
 by auto  
 also have  $\dots \longleftrightarrow \text{match test } r \ i \ i \wedge \text{match test } s \ i \ j \vee$   
 $(\exists k > i. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j)$   
 using *match\_le*[of test r i] *nat\_less\_le* by auto  
 also have  $\dots \longleftrightarrow \text{match test } r \ i \ i \wedge (\bigsqcup t \in \text{lpd test } i \ s. \text{match test } t) \ (i + 1) \ j \vee$   
 $(\exists k > i. (\bigsqcup t \in \text{lpd test } i \ r. \text{match test } t) \ (i + 1) \ k \wedge \text{match test } s \ k \ j)$   
 using *Times.IH(1)* *Times.IH(2)*[OF *Times.prem*s] by metis  
 also have  $\dots \longleftrightarrow \text{match test } r \ i \ i \wedge (\bigsqcup t \in \text{lpd test } i \ s. \text{match test } t) \ (i + 1) \ j \vee$   
 $(\exists k. (\bigsqcup t \in \text{lpd test } i \ r. \text{match test } t) \ (i + 1) \ k \wedge \text{match test } s \ k \ j)$   
 using *Times.prem*s by (intro *disj\_cong*[OF *refl*] *iff\_exI*) (auto dest: *match\_le*)  
 also have  $\dots \longleftrightarrow (\bigsqcup (\text{match test } ' \ \text{lpd test } i \ (Times \ r \ s))) \ (i + 1) \ j$   
 by (force *simp*: *TimesL\_def* *TimesR\_def* *eps\_match*)  
 finally show ?case .

next  
 case (Star r)  
 have  $\exists s \in \text{lpd test } i \ r. (\text{match test } s \ OO \ (\text{match test } r)^{**}) \ (i + 1) \ j$  if  $(\text{match test } r)^{**} \ i \ j$   
 using that *Star.prem*s *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*s **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  $\text{match test } (Times \ r \ s) \ i \ j \longleftrightarrow (\exists k. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j)$   
 by auto  
 also have  $\dots \longleftrightarrow \text{match test } r \ i \ j \wedge \text{match test } s \ j \ j \vee$   
 $(\exists k < j. \text{match test } r \ i \ k \wedge \text{match test } s \ k \ j)$   
 using *match\_le*[of test s \_ j] *nat\_less\_le* by auto  
 also have  $\dots \longleftrightarrow (\bigsqcup t \in \text{rpj test } j \ r. \text{match test } t) \ i \ (j - 1) \wedge \text{match test } s \ j \ j \vee$   
 $(\exists k < j. \text{match test } r \ i \ k \wedge (\bigsqcup t \in \text{rpj test } j \ s. \text{match test } t) \ k \ (j - 1))$   
 using *Times.IH(1)*[OF *Times.prem*s] *Times.IH(2)* by metis  
 also have  $\dots \longleftrightarrow (\bigsqcup t \in \text{rpj test } j \ r. \text{match test } t) \ i \ (j - 1) \wedge \text{match test } s \ j \ j \vee$   
 $(\exists k. \text{match test } r \ i \ k \wedge (\bigsqcup t \in \text{rpj test } j \ s. \text{match test } t) \ k \ (j - 1))$   
 using *Times.prem*s by (intro *disj\_cong*[OF *refl*] *iff\_exI*) (auto dest: *match\_le*)  
 also have  $\dots \longleftrightarrow (\bigsqcup (\text{match test } ' \ \text{rpj test } j \ (Times \ r \ s))) \ i \ (j - 1)$   
 by (force *simp*: *TimesL\_def* *TimesR\_def* *eps\_match*)  
 finally show ?case .

next  
 case (Star r)  
 have  $\exists s \in \text{rpj test } j \ r. ((\text{match test } r)^{**} \ OO \ \text{match test } s) \ i \ (j - 1)$  if  $(\text{match test } r)^{**} \ i \ j$   
 using that *Star.prem*s *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*)<sup>+</sup>

**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*)<sup>+</sup>

**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  $\varphi$ )* = *safe g  $\varphi$*   
| *safe\_regex m g (Plus r s)* =  $((g = \text{Lax} \vee \text{fv\_regex } \text{fv } r = \text{fv\_regex } \text{fv } s) \wedge \text{safe\_regex } m \ g \ r \wedge \text{safe\_regex } m \ g \ s)$   
| *safe\_regex Futu g (Times r s)* =  
 $((g = \text{Lax} \vee \text{fv\_regex } \text{fv } r \subseteq \text{fv\_regex } \text{fv } s) \wedge \text{safe\_regex } \text{Futu } g \ s \wedge \text{safe\_regex } \text{Futu } \text{Lax } r)$   
| *safe\_regex Past g (Times r s)* =  
 $((g = \text{Lax} \vee \text{fv\_regex } \text{fv } s \subseteq \text{fv\_regex } \text{fv } r) \wedge \text{safe\_regex } \text{Past } g \ r \wedge \text{safe\_regex } \text{Past } \text{Lax } s)$   
| *safe\_regex m g (Star r)* =  $((g = \text{Lax} \vee \text{fv\_regex } \text{fv } r = \{\}) \wedge \text{safe\_regex } m \ g \ r)$

**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  $r \implies s \in \text{rpd test } i \ r \implies \text{fv\_regex } fv \ r \subseteq \text{fv\_regex } fv \ s$ 
  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**  $agg\_type = Agg\_Cnt \mid Agg\_Min \mid Agg\_Max \mid Agg\_Sum \mid Agg\_Avg \mid Agg\_Med$   
**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*  $trm\ trm \mid Less\ trm\ trm \mid LessEq\ trm\ trm$   
| *Neg*  $formula \mid Or\ formula\ formula \mid And\ formula\ formula \mid Ands\ formula\ list \mid Exists\ formula$   
| *Agg*  $nat\ agg\_op\ nat\ trm\ formula$   
| *Prev*  $\mathcal{I}\ formula \mid Next\ \mathcal{I}\ formula$   
| *Since*  $formula\ \mathcal{I}\ formula \mid Until\ formula\ \mathcal{I}\ formula$   
| *MatchF*  $\mathcal{I}\ formula\ Regex.regex \mid MatchP\ \mathcal{I}\ formula\ Regex.regex$

**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**  
*nfv*  $\varphi = \text{Max (insert 0 (Suc 'fv } \varphi))$

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

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

**lemma** *nfv\_simps* [*simp*]:  
*nfv* (*Let* *p*  $\varphi$   $\psi$ ) = *nfv*  $\psi$   
*nfv* (*Neg*  $\varphi$ ) = *nfv*  $\varphi$   
*nfv* (*Or*  $\varphi$   $\psi$ ) = *max* (*nfv*  $\varphi$ ) (*nfv*  $\psi$ )  
*nfv* (*And*  $\varphi$   $\psi$ ) = *max* (*nfv*  $\varphi$ ) (*nfv*  $\psi$ )  
*nfv* (*Prev* *I*  $\varphi$ ) = *nfv*  $\varphi$   
*nfv* (*Next* *I*  $\varphi$ ) = *nfv*  $\varphi$   
*nfv* (*Since*  $\varphi$  *I*  $\psi$ ) = *max* (*nfv*  $\varphi$ ) (*nfv*  $\psi$ )  
*nfv* (*Until*  $\varphi$  *I*  $\psi$ ) = *max* (*nfv*  $\varphi$ ) (*nfv*  $\psi$ )  
*nfv* (*MatchP* *I* *r*) = *Regex.nfv\\_regex* *fv* *r*  
*nfv* (*MatchF* *I* *r*) = *Regex.nfv\\_regex* *fv* *r*  
*nfv\_regex* (*Regex.Skip* *n*) = 0  
*nfv\_regex* (*Regex.Test*  $\varphi$ ) = *Max* (*insert* 0 (*Suc* 'fv  $\varphi$ ))  
*nfv\_regex* (*Regex.Plus* *r* *s*) = *max* (*nfv\_regex* *r*) (*nfv\_regex* *s*)  
*nfv\_regex* (*Regex.Times* *r* *s*) = *max* (*nfv\_regex* *r*) (*nfv\_regex* *s*)  
*nfv\_regex* (*Regex.Star* *r*) = *nfv\_regex* *r*  
**unfolding** *nfv\_def* *Regex.nfv\_regex\_def* **by** (*simp\_all* *add: image\_Un Max\_Un* [*symmetric*])

**lemma** *nfv\_Ands* [*simp*]: *nfv* (*Ands* *l*) = *Max* (*insert* 0 (*nfv* 'set *l*))  
**proof** (*induction* *l*)  
**case** *Nil*  
**then show** ?*case* **unfolding** *nfv\_def* **by** *simp*  
**next**  
**case** (*Cons* *a* *l*)  
**have** *fv* (*Ands* (*a* # *l*)) = *fv* *a*  $\cup$  *fv* (*Ands* *l*) **by** *simp*  
**then have** *nfv* (*Ands* (*a* # *l*)) = *max* (*nfv* *a*) (*nfv* (*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  $\infty$ )

**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  $\sigma$  V v i (Ands l)  $\longleftrightarrow$  ( $\forall$   $\varphi$   $\in$  set l. sat  $\sigma$  V v i  $\varphi$ )

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

**proof** (*rule iffI; (elim disjE conjE bexE)?*)

**assume**  $?L$

**then obtain**  $j$  **where**  $j: j \geq i$   $\text{mem } (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$  **and**  $\text{Regex.match } (\text{sat } \sigma\ V\ v)\ r\ i\ j$  **by** *auto*

**then show**  $?R1 \vee ?R2$

**proof** (*cases*  $i < j$ )

**case** *True*

**with**  $\langle \text{Regex.match } (\text{sat } \sigma\ V\ v)\ r\ i\ j \rangle$  *lpd\_match*[*of*  $i\ j\ \text{sat } \sigma\ V\ v\ r$ ]

**obtain**  $s$  **where**  $s \in \text{Regex.lpd } (\text{sat } \sigma\ V\ v)\ i\ r$   $\text{Regex.match } (\text{sat } \sigma\ V\ v)\ s\ (i + 1)\ j$  **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**  $\text{enat } (\Delta\ \sigma\ (i + 1)) \leq \text{right } I$

**moreover**

**fix**  $s$

**assume** [*simp*]:  $s \in \text{Regex.lpd } (\text{sat } \sigma\ V\ v)\ i\ r$  **and**  $\text{sat } \sigma\ V\ v\ (i + 1)\ (\text{MatchF } (\text{subtract } (\Delta\ \sigma\ (i + 1))\ I)\ s)$

**then obtain**  $j$  **where**  $j > i$   $\text{Regex.match } (\text{sat } \sigma\ V\ v)\ s\ (i + 1)\ j$

$\text{mem } (\tau\ \sigma\ j - \tau\ \sigma\ (\text{Suc } i))\ (\text{subtract } (\Delta\ \sigma\ (i + 1))\ I)$  **by** (*auto simp: Suc\_le\_eq*)

**ultimately show**  $?L$

**by** (*cases* *right*  $I$ )

(*auto simp: le\_diff\_conv lpd\_match intro!: exI[of \_ j] bexE[of \_ s]*)

**qed** (*auto simp: eps\_match intro!: exI[of \_ i]*)

**lemma** *sat\_MatchP\_rec*:  $\text{sat } \sigma\ V\ v\ i\ (\text{MatchP } I\ r) \longleftrightarrow \text{mem } 0\ I \wedge \text{Regex.eps } (\text{sat } \sigma\ V\ v)\ i\ r \vee$   
 $i > 0 \wedge \Delta\ \sigma\ i \leq \text{right } I \wedge (\exists s \in \text{Regex.rpd } (\text{sat } \sigma\ V\ v)\ i\ r.\ \text{sat } \sigma\ V\ v\ (i - 1)\ (\text{MatchP } (\text{subtract } (\Delta\ \sigma\ i)\ I)\ s))$

**(is**  $?L \longleftrightarrow ?R1 \vee ?R2$ )

**proof** (*rule iffI; (elim disjE conjE bexE)?*)

**assume**  $?L$

**then obtain**  $j$  **where**  $j: j \leq i$   $\text{mem } (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$  **and**  $\text{Regex.match } (\text{sat } \sigma\ V\ v)\ r\ j\ i$  **by** *auto*

**then show**  $?R1 \vee ?R2$

**proof** (*cases*  $j < i$ )

**case** *True*

**with**  $\langle \text{Regex.match } (\text{sat } \sigma\ V\ v)\ r\ j\ i \rangle$  *rpd\_match*[*of*  $j\ i\ \text{sat } \sigma\ V\ v\ r$ ]

**obtain**  $s$  **where**  $s \in \text{Regex.rpd } (\text{sat } \sigma\ V\ v)\ i\ r$   $\text{Regex.match } (\text{sat } \sigma\ V\ v)\ s\ j\ (i - 1)$  **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**  $\text{enat } (\Delta\ \sigma\ i) \leq \text{right } I$

**moreover**

**fix**  $s$

**assume** [*simp*]:  $s \in \text{Regex.rpd } (\text{sat } \sigma\ V\ v)\ i\ r$  **and**  $\text{sat } \sigma\ V\ v\ (i - 1)\ (\text{MatchP } (\text{subtract } (\Delta\ \sigma\ i)\ I)\ s)$   
 $i > 0$

**then obtain**  $j$  **where**  $j < i$   $\text{Regex.match } (\text{sat } \sigma\ V\ v)\ s\ j\ (i - 1)$

$\text{mem } (\tau\ \sigma\ (i - 1) - \tau\ \sigma\ j)\ (\text{subtract } (\Delta\ \sigma\ i)\ I)$  **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  $\sigma$  V v 0 (Since  $\varphi$  I  $\psi$ )  $\longleftrightarrow$  mem 0 I  $\wedge$  sat  $\sigma$  V v 0  $\psi$   
**by** auto

**lemma** sat\_MatchP\_0: sat  $\sigma$  V v 0 (MatchP I r)  $\longleftrightarrow$  mem 0 I  $\wedge$  Regex.eps (sat  $\sigma$  V v) 0 r  
**by** (auto simp: eps\_match)

**lemma** sat\_Since\_point: sat  $\sigma$  V v i (Since  $\varphi$  I  $\psi$ )  $\implies$   
( $\bigwedge j. j \leq i \implies$  mem ( $\tau \sigma i - \tau \sigma j$ ) I  $\implies$  sat  $\sigma$  V v i (Since  $\varphi$  (point ( $\tau \sigma i - \tau \sigma j$ ))  $\psi$ ))  $\implies$  P)  
 $\implies$  P  
**by** (auto intro: diff\_le\_self)

**lemma** sat\_MatchP\_point: sat  $\sigma$  V v i (MatchP I r)  $\implies$   
( $\bigwedge j. j \leq i \implies$  mem ( $\tau \sigma i - \tau \sigma j$ ) I  $\implies$  sat  $\sigma$  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  $\sigma$  V v i (Since  $\varphi$  (point t)  $\psi$ )  $\implies$  mem t I  $\implies$  sat  $\sigma$  V v i (Since  $\varphi$  I  $\psi$ )  
**by** auto

**lemma** sat\_MatchP\_pointD: sat  $\sigma$  V v i (MatchP (point t) r)  $\implies$  mem t I  $\implies$  sat  $\sigma$  V v i (MatchP I r)  
**by** auto

**lemma** sat\_fv\_cong:  $\forall x \in fv \varphi. v!x = v'!x \implies$  sat  $\sigma$  V v i  $\varphi =$  sat  $\sigma$  V v' i  $\varphi$   
**proof** (induct  $\varphi$  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  $\varphi$   $\psi$ )  
**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  $\sigma$  V v i  $\varphi =$  sat  $\sigma$  V v' i  $\varphi$   
**proof** -  
**fix**  $\varphi$  **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.premis **by** blast  
**then show** sat  $\sigma$  V v i  $\varphi =$  sat  $\sigma$  V v' i  $\varphi$  **using** Ands.hyps  $\langle \varphi \in set l \rangle$  **by** blast  
**qed**  
**then show** ?case **using** sat\_Ands **by** blast  
**next**  
**case** (Exists  $\varphi$ )

```

    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  $\varphi$ ) 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  $\varphi$ ) 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  $\varphi$  = (case  $\varphi$  of

Eq (Var x) (Var y)  $\Rightarrow$  ( $x \notin X \longleftrightarrow y \in X$ )

| Eq (Var x) t  $\Rightarrow$  ( $x \notin X \wedge \text{fv\_trm } t \subseteq X$ )

| Eq t (Var x)  $\Rightarrow$  ( $x \notin X \wedge \text{fv\_trm } t \subseteq X$ )

| \_  $\Rightarrow$  False)

**fun** safe\_formula :: formula  $\Rightarrow$  bool **where**

safe\_formula (Eq t1 t2) = (is\_Const t1  $\wedge$  (is\_Const t2  $\vee$  is\_Var t2))  $\vee$  is\_Var t1  $\wedge$  is\_Const t2)

| safe\_formula (Neg (Eq (Var x) (Var y))) = ( $x = y$ )

| safe\_formula (Less t1 t2) = False

| safe\_formula (LessEq t1 t2) = False

| safe\_formula (Pred e ts) = ( $\forall t \in \text{set } ts. \text{is\_Var } t \vee \text{is\_Const } t$ )

| safe\_formula (Let p  $\varphi$   $\psi$ ) = ( $\{0..<\text{nfv } \varphi\} \subseteq \text{fv } \varphi \wedge \text{safe\_formula } \varphi \wedge \text{safe\_formula } \psi$ )

| safe\_formula (Neg  $\varphi$ ) = ( $\text{fv } \varphi = \{\}$ )  $\wedge$  safe\_formula  $\varphi$

| safe\_formula (Or  $\varphi$   $\psi$ ) = ( $\text{fv } \psi = \text{fv } \varphi \wedge \text{safe\_formula } \varphi \wedge \text{safe\_formula } \psi$ )

| safe\_formula (And  $\varphi$   $\psi$ ) = (safe\_formula  $\varphi \wedge$

(safe\_assignment (fv  $\varphi$ )  $\psi \vee \text{safe\_formula } \psi \vee$

$\text{fv } \psi \subseteq \text{fv } \varphi \wedge (\text{is\_constraint } \psi \vee (\text{case } \psi \text{ of Neg } \psi' \Rightarrow \text{safe\_formula } \psi' \mid \_ \Rightarrow \text{False}))))$ )

| safe\_formula (Ands l) = (let (pos, neg) = partition safe\_formula l in pos  $\neq \{\}$   $\wedge$

$\text{list\_all } \text{safe\_formula } (\text{map } \text{remove\_neg } \text{neg}) \wedge \bigcup (\text{set } (\text{map } \text{fv } \text{neg})) \subseteq \bigcup (\text{set } (\text{map } \text{fv } \text{pos})))$ )

| safe\_formula (Exists  $\varphi$ ) = (safe\_formula  $\varphi$ )

| safe\_formula (Agg y  $\omega$  b f  $\varphi$ ) = (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  $\varphi$ ) = (safe\_formula  $\varphi$ )

| safe\_formula (Next I  $\varphi$ ) = (safe\_formula  $\varphi$ )

| safe\_formula (Since  $\varphi$  I  $\psi$ ) = ( $\text{fv } \varphi \subseteq \text{fv } \psi \wedge$

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

| safe\_formula (Until  $\varphi$  I  $\psi$ ) = ( $\text{fv } \varphi \subseteq \text{fv } \psi \wedge$

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

| 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:

$\text{safe\_regex } m g r \implies \varphi \in \text{Regex.atms } r \implies \text{safe\_formula } \varphi \vee$

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

by (cases g) (auto dest!: safe\_regex\_safe[rotated] split: formula.splits[where formula= $\varphi$ ])

**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  $\varphi$  then  $\{\varphi\}$  else case  $\varphi$  of Neg  $\varphi' \Rightarrow \{\varphi'\} \mid \_ \Rightarrow \{\}$ )

**lemma** atms\_simps[simp]:  
 atms (Regex.Skip n) =  $\{\}$   
 atms (Regex.Test  $\varphi$ ) = (if safe\_formula  $\varphi$  then  $\{\varphi\}$  else case  $\varphi$  of Neg  $\varphi' \Rightarrow \{\varphi'\} \mid \_ \Rightarrow \{\}$ )  
 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  $\varphi$   
**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:  $\bigwedge e ts. \forall t \in \text{set } ts. \text{is\_Var } t \vee \text{is\_Const } t \Longrightarrow P (\text{Pred } e ts)$   
**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:  $\bigwedge l \text{ pos neg}. (\text{pos}, \text{neg}) = \text{partition safe\_formula } l \Longrightarrow \text{pos} \neq [] \Longrightarrow$   
 $\text{list\_all safe\_formula } \text{pos} \Longrightarrow \text{list\_all safe\_formula } (\text{map remove\_neg } \text{neg}) \Longrightarrow$   
 $(\bigcup \varphi \in \text{set } \text{neg}. fv \varphi) \subseteq (\bigcup \varphi \in \text{set } \text{pos}. fv \varphi) \Longrightarrow$   
 $\text{list\_all } P \text{ pos} \Longrightarrow \text{list\_all } P (\text{map remove\_neg } \text{neg}) \Longrightarrow P (\text{Ands } l)$   
**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  $(\text{fv } \varphi) \psi$ 
| (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)  $\psi' \text{ where } \text{fv } \psi \subseteq \text{fv } \varphi \neg \text{safe\_assignment } (\text{fv } \varphi) \psi \neg \text{safe\_formula } \psi \neg \text{is\_constraint } \psi$ 
 $\psi = \text{Neg } \psi' \text{ safe\_formula } \psi'$ 
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:  $(\text{pos}, \text{neg}) = \text{partition safe\_formula } l$  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  $(\text{map remove\_neg } \text{neg})$  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  $P$   $(\text{map remove\_neg } \text{neg})$ 
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  $\varphi$  = {}  $\vee$  ( $\exists x. \varphi = \text{Formula.Eq (Formula.Var } x) (\text{Formula.Var } x)$ )
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  $\varphi e \vee \text{matches } v \psi e$ )
| matches v (And  $\varphi$   $\psi$ ) e = (matches v  $\varphi e \wedge \text{matches } v \psi e$ )
| 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  $\varphi e \vee \text{matches } v \psi e$ )
| matches v (Until  $\varphi$  I  $\psi$ ) e = (matches v  $\varphi e \vee \text{matches } v \psi e$ )
| 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  $(\text{intro } \text{iff\_exI}) (\text{simp add: } \text{fvi\_Suc } \text{nth\_Cons}')$ 
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  $(\text{simp add: } \text{Agg.hyps}[\text{where } v=zs @ v \text{ and } v'=zs @ v]$ 
     $\text{nth\_append } \text{fvi\_iff\_fv}(1)[\text{where } b=b])$ 
  then show  $?case$  by  $\text{auto}$ 
qed  $(\text{auto } 9\ 0 \text{ simp add: } \text{nth\_Cons}'\ \text{fv\_regex\_alt})$ 

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  $(\text{auto simp: } \text{domIff } \text{dest}!: \text{subsetD})$ 
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  $(\text{fastforce intro: } \text{domI})$ 
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  $(\text{elim } \text{subset\_trans}[\text{rotated}]) (\text{auto simp: } \text{set\_eq\_iff})$ 
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: ( $\bigcup x \in (\text{set } (\text{get\_and\_list } \varphi)). \text{fv } b \ x$ ) = fv b  $\varphi$ 
  by (induction  $\varphi$  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  $\varphi$  rule: get_and_list.induct) (simp_all add: safe_neg_def list_all_iff)

```

```

lemma sat_get_and: sat  $\sigma$  V v i  $\varphi \iff \text{list\_all } (\text{sat } \sigma$  V v i) (get_and_list  $\varphi$ )
  by (induction  $\varphi$  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*  $\varphi$  =  $\varphi$

**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*  $\varphi$  *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*  $\varphi$  *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*  $\varphi$  *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*  $\varphi$ ) *simp\_all*

**lemma** *convert\_multiway\_remove\_neg*: *safe\_formula* (*remove\_neg*  $\varphi$ )  $\implies \text{convert\_multiway } (\text{remove\_neg } \varphi)$   
 $= \text{remove\_neg } (\text{convert\_multiway } \varphi)$

**by** (*cases*  $\varphi$ ) (*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*  $\varphi$  *arbitrary*: *b* *rule*: *safe\_formula.induct*)

**case** (9  $\varphi \psi$ )

**then show** *?case* **by** (*cases*  $\psi$ ) (*auto simp*: *fv\_get\_and Un\_commute*)

**next**

**case** (15  $\varphi I \psi$ )

**show** *?case* **proof** (*cases* *safe\_formula*  $\varphi$ )

**case** *True*

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

**next**

**case** *False*

**with** 15.prem*s* **obtain**  $\varphi'$  **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*  $\varphi$ )

**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  $\langle$ safe_formula  $\varphi\rangle \langle$ safe_formula  $\psi\rangle$ 
  by (simp add: safe_get_and)
then have list_all_safe_formula (map remove_neg neg)
proof -
have  $\bigwedge x. x \in \text{set } \text{neg} \implies$  safe_formula (remove_neg  $x$ )
proof -
fix  $x$  assume  $x \in \text{set } \text{neg}$ 
then have  $\neg$  safe_formula  $x$  using posneg by auto
moreover have safe_neg  $x$  using lsafe_neg  $\langle x \in \text{set } \text{neg}\rangle$ 
  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 = Ands (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 = Ands (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 = Ands (?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  $\varphi$  for  $\varphi$  .

**lemma** *sat\_slice\_iff*:  
**assumes**  $v \in S$   
**shows** *Formula*.sat  $\sigma$  V v i  $\varphi \longleftrightarrow$  *Formula*.sat (*Formula\_slicer*.slice  $\varphi$  S  $\sigma$ ) V v i  $\varphi$   
**by** (rule sat\_slice\_strong[OF assms]) auto

**lemma** *Neg\_splits*:  
 $P$  (case  $\varphi$  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  $\varphi$  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  $\varphi$ ; 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`: `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 ( $\lambda b a. \text{if } b \text{ then } a \text{ else } \text{None}$ ) bs as`  
**by** (`induction bs as rule: proj_tuple.induct`) `auto`

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

**lemma** `proj_tuple_join_mask_restrict`: `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  $\Rightarrow$  bool list  $\Rightarrow$  'a tuple  $\Rightarrow$  'a table  $\Rightarrow$  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* ( $\lambda A X$ . table *n A X*  $\wedge$  *wf\_set* *n A*) (*A\_pos* @ *A\_neg*) *L*  
**shows**  $z \in$  *mmulti\_join'* *A\_pos A\_neg L*  $\longleftrightarrow$  *wf\_tuple* *n* ( $\bigcup_{A \in \text{set } A\_pos. A}$ ) *z*  $\wedge$   
*list\_all2* ( $\lambda A X$ . restrict *A z*  $\in$  *X*) *A\_pos* (*take* (*length* *A\_pos*) *L*)  $\wedge$   
*list\_all2* ( $\lambda A X$ . restrict *A z*  $\notin$  *X*) *A\_neg* (*drop* (*length* *A\_pos*) *L*)  
**proof** -  
**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  $\forall (A, X) \in (Q \cup Q\_neg). \text{table } n \ A \ X \wedge \text{wf\_set } n \ A$   
 unfolding Q\_alt Q\_neg\_def using assms(2) by (simp add: zip\_append1 list\_all2\_iff)  
 ultimately have  $z \in ?r \iff \text{wf\_tuple } n \ (\bigcup (A, X) \in Q. A) \ z \wedge$   
 $(\forall (A, X) \in Q. \text{restrict } A \ z \in X) \wedge (\forall (A, X) \in Q\_neg. \text{restrict } A \ z \notin X)$   
 using New\_max.wrapper\_correctness case\_prod\_beta' by blast  
 moreover have  $(\bigcup A \in \text{set } A\_pos. A) = (\bigcup (A, X) \in Q. A)$  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  $\bigwedge z. (\forall (A, X) \in Q. \text{restrict } A \ z \in X) \iff$   
 $\text{list\_all2 } (\lambda A \ X. \text{restrict } A \ z \in X) \ A\_pos \ (\text{take } (\text{length } A\_pos) \ L)$   
 unfolding Q\_alt using assms(2) by (auto simp add: list\_all2\_iff)  
 moreover have  $\bigwedge z. (\forall (A, X) \in Q\_neg. \text{restrict } A \ z \notin X) \iff$   
 $\text{list\_all2 } (\lambda A \ X. \text{restrict } A \ z \notin X) \ A\_neg \ (\text{drop } (\text{length } A\_pos) \ L)$   
 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  $(\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A) \ ((A\_zs @ A\_x \# A\_xs @ A\_y \# A\_ys) @$   
 $A\_neg)$   
 $((zs @ x \# xs @ y \# ys) @ L\_neg)$   
 $\text{length } A\_xs = \text{length } xs \ \text{length } A\_ys = \text{length } ys \ \text{length } A\_zs = \text{length } zs$   
 shows list\_all2  $(\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A)$   
 $((A\_zs @ (A\_x \cup A\_y) \# A\_xs @ A\_ys) @ A\_neg) \ ((zs @ \text{join } x \ \text{True } y \# xs @ ys) @ L\_neg)$   
 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\_append1 dest: list\_all2\_lengthD)  
 then have tabs:  $\text{table } n \ (A\_x \cup A\_y) \ (\text{join } x \ \text{True } y) \ \text{wf\_set } n \ (A\_x \cup A\_y)$   
 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  $(\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A)$   
 $((A\_zs @ (A\_x \cup A\_y) \# A\_xs @ A\_ys) @ A\_neg) \ ((zs @ \text{join } x \ \text{True } y \# xs @ ys) @ L\_neg)$   
 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  $(\lambda A \ X. \text{table } n \ A \ X \wedge \text{wf\_set } n \ A) \ ((A\_zs @ A\_x \# A\_xs @ A\_y \# A\_ys) @$   
 $A\_neg)$   
 $((zs @ x \# xs @ y \# ys) @ L\_neg)$   
 $\text{length } A\_xs = \text{length } xs \ \text{length } A\_ys = \text{length } ys \ \text{length } A\_zs = \text{length } zs$   
 shows mmulti\_join'  $(A\_zs @ A\_x \# A\_xs @ A\_y \# A\_ys) \ A\_neg \ ((zs @ x \# xs @ y \# ys) @$   
 $L\_neg) =$   
 $\text{mmulti\_join}' \ (A\_zs @ (A\_x \cup A\_y) \# A\_xs @ A\_ys) \ A\_neg$   
 $((zs @ \text{join } x \ \text{True } y \# xs @ ys) @ L\_neg)$

**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 have list_all2': 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
have res: ∧z Z. wf_tuple n Z z ⇒ A_x ∪ A_y ⊆ Z ⇒
  restrict (A_x ∪ A_y) z ∈ join x True y ⇔ restrict A_x z ∈ x ∧ restrict A_y z ∈ y
  using join_restrict[of x n A_x y A_y True] wf_tuple_restrict_simple[of n _ _ A_x ∪ 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 ∈ 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 (∪(set (A_zs @ A_x # A_xs @ A_y # A_ys))) z
    list_all2 (λA. (∈) (restrict A z)) A_zs zs
    restrict A_x z ∈ x
    list_all2 (λA. (∈) (restrict A z)) A_ys ys
    restrict A_y z ∈ y
    list_all2 (λA. (∈) (restrict A z)) A_xs xs
    list_all2 (λA. (∉) (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 ∈ mmulti_join' (A_zs @ (A_x ∪ 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 ∈ mmulti_join' (A_zs @ (A_x ∪ 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 (∪(set (A_zs @ A_x # A_xs @ A_y # A_ys))) z
    list_all2 (λA. (∈) (restrict A z)) A_zs zs
    restrict (A_x ∪ A_y) z ∈ join x True y
    list_all2 (λA. (∈) (restrict A z)) A_ys ys
    list_all2 (λA. (∈) (restrict A z)) A_xs xs
    list_all2 (λA. (∉) (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 ∈ 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'* ( $(A\_zs @ A\_x \# A\_xs) (A\_ws @ A\_y \# A\_ys) ((zs @ x \# xs) @ (ws @ y \# ys))$ ) =  
*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  $\text{mmulti\_join}'\_correct[OF\_list\_all2', of z]$   $res$ 
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  $\text{mmulti\_join}'\_correct[OF\_list\_all2', of z]$ 
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  $\text{mmulti\_join}'\_correct[OF\_assms(1), of z]$   $res$ 
by ( $\text{auto simp add: assms list\_all2\_appendI Un\_assoc simp del: mmulti\_join}'.simps$ 
  dest: list_all2_lengthD)
qed
qed

fun  $\text{find\_sub\_in} :: 'a \text{ set} \Rightarrow 'a \text{ set list} \Rightarrow \text{bool} \Rightarrow$ 
  ( $'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list}$ ) 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  $\text{find\_sub\_True} :: 'a \text{ set list} \Rightarrow$ 
  ( $'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list}$ ) 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  $\text{find\_sub\_False} :: 'a \text{ set list} \Rightarrow 'a \text{ set list} \Rightarrow$ 
  ( $'a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list}$ )  $\times ('a \text{ set list} \times 'a \text{ set} \times 'a \text{ set list})$  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* ::  $\text{nat set list} \Rightarrow 'a \text{ table list} \Rightarrow \text{nat set list} \Rightarrow 'a \text{ table list} \Rightarrow$

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

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

$'a \text{ table list} \times 'a \text{ table} \times 'a \text{ table list}))$  **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.prem(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.prem(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  $y \in \text{set negm } x = \text{remove\_neg } y$  by auto
  then have  $y \in \text{set negs}$  using  $\langle \text{set negm} \subseteq \text{set negs} \rangle$  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]  $\langle x = \text{remove\_neg } y \rangle$   $\langle y \in \text{set negs} \rangle$  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)  $\langle y \in \text{set negs} \rangle$  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  $(\bigcup_{x \in \text{set negm. } fv } x) \subseteq (\bigcup_{x \in \text{set posm. } fv } x)$ 
  using  $\langle \bigcup (fv \text{ ' set negs}) \subseteq \bigcup (fv \text{ ' set poss}) \rangle$   $\langle \text{set poss} \subseteq \text{set posm} \rangle$   $\langle \text{set negm} \subseteq \text{set negs} \rangle$ 
  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) ==> set ys = atms r ^ 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  $\varphi$ )  $\subseteq$  {MTestPos  $\varphi$ }  
 RPDi i (MTestNeg  $\varphi$ )  $\subseteq$  {MTestNeg  $\varphi$ }  
 by (induct i) auto

**lemma** RPDs\_Test:

RPDs (MSkip 0)  $\subseteq$  {MSkip 0}  
 RPDs (MTestPos  $\varphi$ )  $\subseteq$  {MTestPos  $\varphi$ }  
 RPDs (MTestNeg  $\varphi$ )  $\subseteq$  {MTestNeg  $\varphi$ }  
 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 ( $\bigcup_{j \leq i} \text{RPDi } j \text{ s}$ )  $\cup$  ( $\bigcup_{j \leq i} \text{RPDi } j \text{ r}$ )

**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:  $j \leq i \implies \text{RPDi } j \text{ (MStar r)} \subseteq \{\text{MStar r}\} \cup \text{MTimesL (MStar r) } (\bigcup_{j \leq i} \text{RPDi } j \text{ r})$

**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 = saturate (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 = 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 $\delta$ aux = (ts  $\times$  'a table list  $\times$  '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  $\times$  Formula.trm  
 | MAndRel ('msaux, 'muaux) mformula Formula.trm  $\times$  bool  $\times$  mconstraint  $\times$  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 $\delta$ aux  
 | MMatchF  $\mathcal{I}$  mregex mregex list ('msaux, 'muaux) mformula list event\_data mbufn ts list event\_data  
 ml $\delta$ 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')  $\implies$  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')  $\implies$ 
  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')  $\implies$ 
  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  $\Rightarrow$  ts  $\Rightarrow$  'muaux  $\Rightarrow$  event_data muaux  $\Rightarrow$  bool
  and init_muaux :: args  $\Rightarrow$  'muaux
  and add_new_muaux :: args  $\Rightarrow$  event_data table  $\Rightarrow$  event_data table  $\Rightarrow$  ts  $\Rightarrow$  'muaux  $\Rightarrow$  'muaux
  and length_muaux :: args  $\Rightarrow$  'muaux  $\Rightarrow$  nat
  and eval_muaux :: args  $\Rightarrow$  ts  $\Rightarrow$  'muaux  $\Rightarrow$  event_data table list  $\times$  'muaux
  assumes valid_init_muaux: L  $\subseteq$  R  $\implies$ 
  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  $\implies$ 
  table (args_n args) (args_L args) rel1  $\implies$ 
  table (args_n args) (args_R args) rel2  $\implies$ 
  nt  $\geq$  cur  $\implies$ 
  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  $\implies$  length_muaux args aux = length
auxlist
  assumes valid_eval_muaux: valid_muaux args cur aux auxlist  $\implies$  nt  $\geq$  cur  $\implies$ 
  eval_muaux args nt aux = (res, aux')  $\implies$  eval_until (args_ivl args) nt auxlist = (res', auxlist')  $\implies$ 
  res = res'  $\wedge$  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  $\Rightarrow$  ts  $\Rightarrow$  'msaux  $\Rightarrow$  event_data msaux  $\Rightarrow$  bool
  and init_msaux :: args  $\Rightarrow$  'msaux
  and add_new_ts_msaux :: args  $\Rightarrow$  ts  $\Rightarrow$  'msaux  $\Rightarrow$  'msaux
  and join_msaux :: args  $\Rightarrow$  event_data table  $\Rightarrow$  'msaux  $\Rightarrow$  'msaux
  and add_new_table_msaux :: args  $\Rightarrow$  event_data table  $\Rightarrow$  'msaux  $\Rightarrow$  'msaux
  and result_msaux :: args  $\Rightarrow$  'msaux  $\Rightarrow$  event_data table
  and valid_muaux :: args  $\Rightarrow$  ts  $\Rightarrow$  'muaux  $\Rightarrow$  event_data muaux  $\Rightarrow$  bool
  and init_muaux :: args  $\Rightarrow$  'muaux
  and add_new_muaux :: args  $\Rightarrow$  event_data table  $\Rightarrow$  event_data table  $\Rightarrow$  ts  $\Rightarrow$  'muaux  $\Rightarrow$  'muaux
  and length_muaux :: args  $\Rightarrow$  'muaux  $\Rightarrow$  nat
  and eval_muaux :: args  $\Rightarrow$  nat  $\Rightarrow$  'muaux  $\Rightarrow$  event_data table list  $\times$  'muaux

fun split_assignment :: nat set  $\Rightarrow$  Formula.formula  $\Rightarrow$  nat  $\times$  Formula.trm where
  split_assignment X (Formula.Eq t1 t2) = (case (t1, t2) of
    (Formula.Var x, Formula.Var y)  $\Rightarrow$  if x  $\in$  X then (y, t1) else (x, t2)
  | (Formula.Var x, _)  $\Rightarrow$  (x, t2)
  | (_, Formula.Var y)  $\Rightarrow$  (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*  $\varphi$  = (*let* *n* = *Formula.nfv*  $\varphi$  in ( $\downarrow$ *mstate\_i* = 0, *mstate\_m* = *minit0* *n*  $\varphi$ , *mstate\_n* = *n*))

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

**fun** *mprev\_next* :: *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* ( $\lambda$ *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* ( $\lambda$ (\_, \_, *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 φ first nts) =
  (let (xs, φ) = meval n t db φ;
      (xs, first) = (case (xs, first) of (_ # xs, True) ⇒ (xs, False) | a ⇒ a);
      (zs, _, nts) = mprev_next I xs (nts @ [t])
      in (zs, MNext I φ first nts))
| meval n t db (MSince args φ ψ buf nts aux) =
  (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
      ((zs, aux), buf, nts) = mbuf2t_take (λr1 r2 t (zs, aux).
        let (z, aux) = update_since args r1 r2 t aux
            in (zs @ [z], aux) (mbuf2_add xs ys buf) (nts @ [t]))
      in (zs, MSince args φ ψ buf nts aux))
| meval n t db (MUntil args φ ψ buf nts aux) =
  (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
      (aux, buf, nts) = mbuf2t_take (add_new_muaux args) aux (mbuf2_add xs ys buf) (nts @ [t]);
      (zs, aux) = eval_muaux args (case nts of [] ⇒ t | nt # _ ⇒ nt) aux
      in (zs, MUntil args φ ψ buf nts aux))
| meval n t db (MMatchP I mr mrs φs buf nts aux) =
  (let (xss, φs) = map_split id (map (meval n t db) φs);
      ((zs, aux), buf, nts) = mbufnt_take (λrels t (zs, aux).
        let (z, aux) = update_matchP n I mr mrs rels t aux
            in (zs @ [z], aux) (mbufn_add xss buf) (nts @ [t]))
      in (zs, MMatchP I mr mrs φs buf nts aux))
| meval n t db (MMatchF I mr mrs φs buf nts aux) =
  (let (xss, φs) = map_split id (map (meval n t db) φs);
      (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 [] ⇒ t | nt # _ ⇒ nt) aux
      in (zs, MMatchF I mr mrs φs buf nts aux))

```

**definition** (in *maux*) *mstep* :: *Formula.database* × *ts* ⇒ ('*msaux*', '*muaux*') *mstate* ⇒ (*nat* × *event\_data table*) *list* × ('*msaux*', '*muaux*') *mstate* **where**

```

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

```

## 6.4 Verdict delay

**context fixes**  $\sigma$  :: *Formula.trace* **begin**

```

fun progress :: (Formula.name → nat) ⇒ Formula.formula ⇒ nat ⇒ nat where
  progress P (Formula.Pred e ts) j = (case P e of None ⇒ j | Some k ⇒ k)
| progress P (Formula.Let p  $\varphi$   $\psi$ ) j = progress (P(p ⇨ 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 (λ $\varphi$ . 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. ∀ k. k < j ∧ k ≤ min (progress P  $\varphi$  j) (progress P  $\psi$  j) → τ σ i + right I ≥ τ σ 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**  $\text{progress.simps}[simp \ del]$

**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  $i \leq \text{progress } \sigma (\text{max\_mapping } P1 P2)$  (Formula.Or  $\varphi1 \varphi2$ ) (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: dom P1 = S 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: dom P2 = S range_mapping i j2 P2  $i \leq \text{progress } \sigma P2 \varphi2 j2$ 
  using And(2)[of S i] by auto
ultimately have  $i \leq \text{progress } \sigma (\text{max\_mapping } P1 P2)$  (Formula.And  $\varphi1 \varphi2$ ) (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.prem 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.prem obtain P j where dom P = S 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.prem 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:  $\forall \varphi \in \text{set } l. ?P (Pjf \varphi) \varphi ..$ 
  define Pf where Pf = fst o Pjf
  define jf where jf = snd o Pjf
  have *: dom (Pf  $\varphi$ ) = S 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=MAX  $\varphi \in \text{set } l. \text{jf } \varphi$ ] exI[where x=Max_mapping (Pf '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)

```

```

^
  Suc i' ≤ progress σ (fst Pj) φ (snd Pj)) auto
let ?P = Max_mapping (pickP ' regex.atms r) let ?j = Max (pickj ' regex.atms r)
from pick[OF φ] False φ have Suc i' ≤ ?j
by (intro order_trans[OF pick[THEN conjunct2], THEN conjunct2], OF φ] 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 τ_mono[of _ ?j σ] less_τD[of σ 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 φ ⇒ ∃j. i ≤ progress σ Map.empty φ j
  using progress_ge_gen[of φ {} i σ]
  by auto

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

lemma progress_time_conv:
  assumes ∀i<j. τ σ i = τ σ' i
  shows progress σ P φ j = progress σ' P φ j
  using assms proof (induction φ arbitrary: P)
  case (Until φ1 I φ2)
  have *: i ≤ j - 1 ↔ i < j if j ≠ 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 τ_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 ≤ j - 1 ↔ i < j if j ≠ 0 for i
    using that by auto
  with MatchF show ?case using τ_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  $\sigma \ V \ v \ i \ \varphi \longleftrightarrow \text{Formula.sat } \sigma' \ V' \ v \ i \ \varphi$ 
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 Some  $\langle \text{dom } V = \text{dom } V' \rangle$  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
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

```

```

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]  $\tau$ _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  $\tau$ _mono, rotated])
next
case (RL j)
with Until(1)[OF 11] Until(2)[OF 21]  $\tau$ _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  $\tau$ _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  $\tau$ _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*  $\pi$   $\sigma$  **and** *prefix\_of*  $\pi$   $\sigma'$   
**shows**  $i < \text{progress } \sigma \text{ Map.empty } \varphi (\text{plen } \pi) \implies$   
 $\text{Formula.sat } \sigma \text{ Map.empty } v \ i \ \varphi \longleftrightarrow \text{Formula.sat } \sigma' \text{ Map.empty } v \ i \ \varphi$   
**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*  $\varphi$ ) *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*  $\varphi$  *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*  $\varphi$  *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  $\_ \_ \text{Min}$ ] *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  $\_ \_ \text{Min}$ ] *arg\_cong*[of  $\_ \_ \text{Inf}$ ] *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**  $future\_bounded\_mfodl \subseteq sliceable\_timed\_progress Formula.nfv \varphi Formula.fv \varphi relevant\_events$

$\varphi$

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

**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**  $\sigma$  **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  $\sigma x$ )

**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,  $\varphi$ s)  $\Rightarrow$   
 ( $\forall$  ms  $\in$  RPDs mr. qtable n (Formula.fv\_regex r) (mem\_restr R) ( $\lambda$ v. Formula.sat  $\sigma$  V (map the v)  
 (ne-1)  
 (Formula.MatchP (point ( $\tau$   $\sigma$  (ne-1) - t)) (from\_mregex ms  $\varphi$ s)))  
 (lookup X ms)))  $\wedge$   
 ( $\forall$  t. 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)  $\longrightarrow$   
 ( $\exists$  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  $\sigma$  V UNIV args  $\varphi$   $\psi$  aux ne  $\longleftrightarrow$  Formula.fv  $\varphi$   $\subseteq$  Formula.fv  $\psi$   $\wedge$  ( $\exists$  cur auxlist.  
 valid\_msaux args cur aux auxlist  $\wedge$   
 cur = (if ne = 0 then 0 else  $\tau$   $\sigma$  (ne - 1))  $\wedge$   
 sorted\_wrt ( $\lambda$ x y. fst x > fst y) auxlist  $\wedge$   
 ( $\forall$  t X. (t, X)  $\in$  set auxlist  $\longrightarrow$  ne  $\neq$  0  $\wedge$  t  $\leq$   $\tau$   $\sigma$  (ne - 1)  $\wedge$   $\tau$   $\sigma$  (ne - 1) - t  $\leq$  right (args\_ivl  
 args)  $\wedge$  ( $\exists$  i.  $\tau$   $\sigma$  i = t)  $\wedge$   
 wf\_table (args\_n args) (Formula.fv  $\psi$ )  
 ( $\lambda$ v. Formula.sat  $\sigma$  V (map the v) (ne - 1) (Sincep (args\_pos args)  $\varphi$  (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  $\leq$  right (args\_ivl args)  $\wedge$  ( $\exists$  i.  $\tau$   $\sigma$  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  $\sigma$  V n R pos  $\varphi$  I  $\psi$  auxlist ne  $\longleftrightarrow$   
 list\_all2 ( $\lambda$ x i. case x of (t, r1, r2)  $\Rightarrow$  t =  $\tau$   $\sigma$  i  $\wedge$   
 qtable n (Formula.fv  $\varphi$ ) (mem\_restr R) ( $\lambda$ v. if pos then ( $\forall$  k  $\in$  {i..
 $\sigma$  V (map the v) k  $\varphi$ )  
 else ( $\exists$  k  $\in$  {i..\sigma V (map the v) k  $\varphi$ )) r1  $\wedge$   
 qtable n (Formula.fv  $\psi$ ) (mem\_restr R) ( $\lambda$ v. ( $\exists$  j. i  $\leq$  j  $\wedge$  j < ne + length auxlist  $\wedge$  mem ( $\tau$   $\sigma$  j -  
 $\tau$   $\sigma$  i) I)  $\wedge$   
 Formula.sat  $\sigma$  V (map the v) j  $\psi$   $\wedge$   
 ( $\forall$  k  $\in$  {i..\sigma V (map the v) k  $\varphi$  else  $\neg$  Formula.sat  $\sigma$  V (map the v)  
 k  $\varphi$ )) r2)  
 auxlist [ne..

**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  $\sigma$  V R args  $\varphi$   $\psi$  aux ne  $\longleftrightarrow$  Formula.fv  $\varphi$   $\subseteq$  Formula.fv  $\psi$   $\wedge$   
 ( $\exists$  cur auxlist. valid\_muaux args cur aux auxlist  $\wedge$   
 cur = (if ne + length auxlist = 0 then 0 else  $\tau$   $\sigma$  (ne + length auxlist - 1))  $\wedge$   
 wf\_until\_auxlist  $\sigma$  V (args\_n args) R (args\_pos args)  $\varphi$  (args\_ivl args)  $\psi$  auxlist ne)

**lemma** (in muaux) wf\_until\_aux\_UNIV\_alt:

wf\_until\_aux  $\sigma$  V UNIV args  $\varphi$   $\psi$  aux ne  $\longleftrightarrow$  Formula.fv  $\varphi$   $\subseteq$  Formula.fv  $\psi$   $\wedge$   
 ( $\exists$  cur auxlist. valid\_muaux args cur aux auxlist  $\wedge$   
 cur = (if ne + length auxlist = 0 then 0 else  $\tau$   $\sigma$  (ne + length auxlist - 1))  $\wedge$   
 list\_all2 ( $\lambda$ x i. case x of (t, r1, r2)  $\Rightarrow$  t =  $\tau$   $\sigma$  i  $\wedge$   
 wf\_table (args\_n args) (Formula.fv  $\varphi$ ) ( $\lambda$ v. if (args\_pos args)  
 then ( $\forall$  k  $\in$  {i..\sigma V (map the v) k  $\varphi$ )  
 else ( $\exists$  k  $\in$  {i..\sigma V (map the v) k  $\varphi$ )) r1  $\wedge$   
 wf\_table (args\_n args) (Formula.fv  $\psi$ ) ( $\lambda$ v. ( $\exists$  j. i  $\leq$  j  $\wedge$  j < ne + length auxlist  $\wedge$  mem ( $\tau$   $\sigma$  j -  $\tau$   
 $\sigma$  i) (args\_ivl args)  $\wedge$   
 Formula.sat  $\sigma$  V (map the v) j  $\psi$   $\wedge$

$(\forall k \in \{i..<j\}). \text{if } (\text{args\_pos } \text{args}) \text{ then } \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ k \ \varphi \ \text{else } \neg \text{Formula.sat } \sigma \ V$   
 $(\text{map the } v) \ k \ \varphi) \ r2)$   
 $\text{auxlist } [ne..<ne+\text{length } \text{auxlist}]$   
**unfolding**  $\text{wf\_until\_aux\_def } \text{wf\_until\_auxlist\_def } \text{qtable\_mem\_restr\_UNIV } ..$

**definition**  $\text{wf\_matchF\_aux} :: \text{Formula.trace} \Rightarrow \_ \Rightarrow \text{nat} \Rightarrow \text{event\_data list set} \Rightarrow$   
 $\mathcal{I} \Rightarrow \text{Formula.formula } \text{Regex.regex} \Rightarrow \text{event\_data } m\delta\text{aux} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
 $\text{wf\_matchF\_aux } \sigma \ V \ n \ R \ I \ r \ \text{aux} \ ne \ k \ \longleftrightarrow (\text{case to\_mregex } r \ \text{of } (mr, \varphi s) \Rightarrow$   
 $\text{list\_all2 } (\lambda x \ i. \ \text{case } x \ \text{of } (t, \text{rels}, \text{rel}) \Rightarrow t = \tau \ \sigma \ i \wedge$   
 $\text{list\_all2 } (\lambda \varphi. \ \text{qtable } n \ (\text{Formula.fv } \varphi) \ (\text{mem\_restr } R) \ (\lambda v.$   
 $\text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi)) \ \varphi s \ \text{rels } \wedge$   
 $\text{qtable } n \ (\text{Formula.fv\_regex } r) \ (\text{mem\_restr } R) \ (\lambda v. (\exists j. \ i \leq j \wedge j < ne + \text{length } \text{aux} + k \wedge \text{mem}$   
 $(\tau \ \sigma \ j - \tau \ \sigma \ i) \ I \wedge$   
 $\text{Regex.match } (\text{Formula.sat } \sigma \ V \ (\text{map the } v)) \ r \ i \ j)) \ \text{rel})$   
 $\text{aux } [ne..<ne+\text{length } \text{aux}]$

**definition**  $\text{wf\_matchF\_invar}$  **where**  
 $\text{wf\_matchF\_invar } \sigma \ V \ n \ R \ I \ r \ st \ i =$   
 $(\text{case } st \ \text{of } (\text{aux}, Y) \Rightarrow \text{aux} \neq [] \wedge \text{wf\_matchF\_aux } \sigma \ V \ n \ R \ I \ r \ \text{aux} \ i \ 0 \wedge$   
 $(\text{case to\_mregex } r \ \text{of } (mr, \varphi s) \Rightarrow \forall ms \in \text{LPDs } mr.$   
 $\text{qtable } n \ (\text{Formula.fv\_regex } r) \ (\text{mem\_restr } R) \ (\lambda v.$   
 $\text{Regex.match } (\text{Formula.sat } \sigma \ V \ (\text{map the } v)) \ (\text{from\_mregex } ms \ \varphi s) \ i \ (i + \text{length } \text{aux} - 1)) \ (\text{lookup}$   
 $Y \ ms)))$

**definition**  $\text{lift\_envs'}$  **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**  $\sigma \ j$  **where**  
 $\text{Eq: is\_simple\_eq } t1 \ t2 \ \Longrightarrow$   
 $\forall x \in \text{Formula.fv\_trm } t1. \ x < n \ \Longrightarrow \forall x \in \text{Formula.fv\_trm } t2. \ x < n \ \Longrightarrow$   
 $\text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ (\text{MRel } (\text{eq\_rel } n \ t1 \ t2)) \ (\text{Formula.Eq } t1 \ t2)$   
 $|\ \text{neg\_Var: } x < n \ \Longrightarrow$   
 $\text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ (\text{MRel } \text{empty\_table}) \ (\text{Formula.Neg } (\text{Formula.Eq } (\text{Formula.Var } x) \ (\text{Formula.Var}$   
 $x))))$   
 $|\ \text{Pred: } \forall x \in \text{Formula.fv} \ (\text{Formula.Pred } e \ ts). \ x < n \ \Longrightarrow$   
 $\forall t \in \text{set } ts. \ \text{Formula.is\_Var } t \vee \text{Formula.is\_Const } t \ \Longrightarrow$   
 $\text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ (\text{MPred } e \ ts) \ (\text{Formula.Pred } e \ ts)$   
 $|\ \text{Let: } \text{wf\_mformula } \sigma \ j \ P \ V \ m \ \text{UNIV } \varphi \ \varphi' \ \Longrightarrow$   
 $\text{wf\_mformula } \sigma \ j \ (P(p \mapsto \text{progress } \sigma \ P \ \varphi' \ j))$   
 $\quad (V(p \mapsto \lambda i. \ \{v. \ \text{length } v = m \wedge \text{Formula.sat } \sigma \ V \ v \ i \ \varphi')) \ n \ R \ \psi \ \psi' \ \Longrightarrow$   
 $\{0..<m\} \subseteq \text{Formula.fv } \varphi' \ \Longrightarrow b \leq m \ \Longrightarrow m = \text{Formula.nfv } \varphi' \ \Longrightarrow$   
 $\text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ (\text{MLet } p \ m \ \varphi \ \psi) \ (\text{Formula.Let } p \ \varphi' \ \psi')$   
 $|\ \text{And: } \text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \ \Longrightarrow \text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ \psi \ \psi' \ \Longrightarrow$   
 $\text{if } \text{pos} \ \text{then } \chi = \text{Formula.And } \varphi' \ \psi'$   
 $\quad \text{else } \chi = \text{Formula.And } \varphi' \ (\text{Formula.Neg } \psi') \wedge \text{Formula.fv } \psi' \subseteq \text{Formula.fv } \varphi' \ \Longrightarrow$   
 $\text{wf\_mbuf2' } \sigma \ P \ V \ j \ n \ R \ \varphi' \ \psi' \ \text{buf} \ \Longrightarrow$   
 $\text{wf\_mformula } \sigma \ j \ P \ V \ n \ R \ (\text{MAnd } (\text{fv } \varphi') \ \varphi \ \text{pos } (\text{fv } \psi') \ \psi \ \text{buf}) \ \chi$

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

$| \text{AndRel: } wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$   
 $\psi' = \text{formula\_of\_constraint } \text{conf} \implies$   
 $(\text{let } (t1, \_, \_, t2) = \text{conf} \ \text{in } \text{Formula.fv\_trm } t1 \cup \text{Formula.fv\_trm } t2 \subseteq \text{Formula.fv } \varphi') \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MAndRel } \varphi \ \text{conf}) \ (\text{Formula.And } \varphi' \ \psi')$

$| \text{Ands: } list\_all2 \ (\lambda\varphi \ \varphi'. wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi') \ l \ (l\_pos \ @ \ \text{map } \text{remove\_neg } \ l\_neg) \implies$   
 $wf\_mbufn \ (\text{progress } \sigma \ P \ (\text{Formula.Ands } l') \ j) \ (\text{map } (\lambda\psi. \text{progress } \sigma \ P \ \psi \ j) \ (l\_pos \ @ \ \text{map } \text{remove\_neg} \ l\_neg)) \ (\text{map } (\lambda\psi \ i.$   
 $\quad \text{qtable } n \ (\text{Formula.fv } \psi) \ (\text{mem\_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \psi)) \ (l\_pos \ @ \ \text{map}$   
 $\text{remove\_neg } \ l\_neg)) \ \text{buf} \implies$   
 $(l\_pos, l\_neg) = \text{partition } \text{safe\_formula } l' \implies$   
 $l\_pos \neq [] \implies$   
 $list\_all \ \text{safe\_formula} \ (\text{map } \text{remove\_neg } \ l\_neg) \implies$   
 $A\_pos = \text{map } \text{fv} \ l\_pos \implies$   
 $A\_neg = \text{map } \text{fv} \ l\_neg \implies$   
 $\bigcup(\text{set } A\_neg) \subseteq \bigcup(\text{set } A\_pos) \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MAnds } A\_pos \ A\_neg \ l \ \text{buf}) \ (\text{Formula.Ands } l')$

$| \text{Or: } wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \psi \ \psi' \implies$   
 $\text{Formula.fv } \varphi' = \text{Formula.fv } \psi' \implies$   
 $wf\_mbuf2' \ \sigma \ P \ V \ j \ n \ R \ \varphi' \ \psi' \ \text{buf} \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MOr } \varphi \ \psi \ \text{buf}) \ (\text{Formula.Or } \varphi' \ \psi')$

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

$| \text{Exists: } wf\_mformula \ \sigma \ j \ P \ V \ (\text{Suc } n) \ (\text{lift\_envs } R) \ \varphi \ \varphi' \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MExists } \varphi) \ (\text{Formula.Exists } \varphi')$

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

$| \text{Prev: } wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$   
 $\text{first} \longleftrightarrow j = 0 \implies$   
 $list\_all2 \ (\lambda i. \text{qtable } n \ (\text{Formula.fv } \varphi') \ (\text{mem\_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ \varphi'))$   
 $\quad [\text{min } (\text{progress } \sigma \ P \ \varphi' \ j) \ (j-1)..<\text{progress } \sigma \ P \ \varphi' \ j] \ \text{buf} \implies$   
 $list\_all2 \ (\lambda i \ t. \ t = \tau \ \sigma \ i) \ [\text{min } (\text{progress } \sigma \ P \ \varphi' \ j) \ (j-1)..<j] \ \text{nts} \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MPrev } I \ \varphi \ \text{first} \ \text{buf} \ \text{nts}) \ (\text{Formula.Prev } I \ \varphi')$

$| \text{Next: } wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies$   
 $\text{first} \longleftrightarrow \text{progress } \sigma \ P \ \varphi' \ j = 0 \implies$   
 $list\_all2 \ (\lambda i \ t. \ t = \tau \ \sigma \ i) \ [\text{progress } \sigma \ P \ \varphi' \ j - 1..<j] \ \text{nts} \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MNext } I \ \varphi \ \text{first} \ \text{nts}) \ (\text{Formula.Next } I \ \varphi')$

$| \text{Since: } wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \varphi \ \varphi' \implies wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ \psi \ \psi' \implies$   
 $\text{if } \text{args\_pos } \ \text{args} \ \text{then } \varphi'' = \varphi' \ \text{else } \varphi'' = \text{Formula.Neg } \varphi' \implies$   
 $\text{safe\_formula } \varphi'' = \text{args\_pos } \ \text{args} \implies$   
 $\text{args\_ivl } \ \text{args} = I \implies$   
 $\text{args\_n } \ \text{args} = n \implies$   
 $\text{args\_L } \ \text{args} = \text{Formula.fv } \varphi' \implies$   
 $\text{args\_R } \ \text{args} = \text{Formula.fv } \psi' \implies$   
 $\text{Formula.fv } \varphi' \subseteq \text{Formula.fv } \psi' \implies$   
 $wf\_mbuf2' \ \sigma \ P \ V \ j \ n \ R \ \varphi' \ \psi' \ \text{buf} \implies$   
 $wf\_ts \ \sigma \ P \ j \ \varphi' \ \psi' \ \text{nts} \implies$   
 $wf\_since\_aux \ \sigma \ V \ R \ \text{args } \varphi' \ \psi' \ \text{aux} \ (\text{progress } \sigma \ P \ (\text{Formula.Since } \varphi'' \ I \ \psi') \ j) \implies$   
 $wf\_mformula \ \sigma \ j \ P \ V \ n \ R \ (\text{MSince } \text{args } \varphi \ \psi \ \text{buf} \ \text{nts} \ \text{aux}) \ (\text{Formula.Since } \varphi'' \ I \ \psi')$

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

$if\ args\_pos\ args\ then\ \varphi'' = \varphi' \ else\ \varphi'' = Formula.Neg\ \varphi' \implies$   
 $safe\_formula\ \varphi'' = args\_pos\ args \implies$   
 $args\_ivl\ args = I \implies$   
 $args\_n\ args = n \implies$   
 $args\_L\ args = Formula.fv\ \varphi' \implies$   
 $args\_R\ args = Formula.fv\ \psi' \implies$   
 $Formula.fv\ \varphi' \subseteq Formula.fv\ \psi' \implies$   
 $wf\_mbuf2'\ \sigma\ P\ V\ j\ n\ R\ \varphi'\ \psi'\ buf \implies$   
 $wf\_ts\ \sigma\ P\ j\ \varphi'\ \psi'\ nts \implies$   
 $wf\_until\_aux\ \sigma\ V\ R\ args\ \varphi'\ \psi'\ aux\ (progress\ \sigma\ P\ (Formula.Until\ \varphi''\ I\ \psi')\ j) \implies$   
 $progress\ \sigma\ P\ (Formula.Until\ \varphi''\ I\ \psi')\ j + length\_muaux\ args\ aux = \min\ (progress\ \sigma\ P\ \varphi'\ j)\ (progress\ \sigma\ P\ \psi'\ j) \implies$   
 $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MUntil\ args\ \varphi\ \psi\ buf\ nts\ aux)\ (Formula.Until\ \varphi''\ I\ \psi')$   
 $| MatchP: (case\ to\_mregex\ r\ of\ (mr',\ \varphi s') \implies$   
 $list\_all2\ (wf\_mformula\ \sigma\ j\ P\ V\ n\ R)\ \varphi s\ \varphi s' \wedge mr = mr') \implies$   
 $mrs = sorted\_list\_of\_set\ (RPDs\ mr) \implies$   
 $safe\_regex\ Past\ Strict\ r \implies$   
 $wf\_mbufn'\ \sigma\ P\ V\ j\ n\ R\ r\ buf \implies$   
 $wf\_ts\_regex\ \sigma\ P\ j\ r\ nts \implies$   
 $wf\_matchP\_aux\ \sigma\ V\ n\ R\ I\ r\ aux\ (progress\ \sigma\ P\ (Formula.MatchP\ I\ r)\ j) \implies$   
 $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MMatchP\ I\ mr\ mrs\ \varphi s\ buf\ nts\ aux)\ (Formula.MatchP\ I\ r)$   
 $| MatchF: (case\ to\_mregex\ r\ of\ (mr',\ \varphi s') \implies$   
 $list\_all2\ (wf\_mformula\ \sigma\ j\ P\ V\ n\ R)\ \varphi s\ \varphi s' \wedge mr = mr') \implies$   
 $mrs = sorted\_list\_of\_set\ (LPDs\ mr) \implies$   
 $safe\_regex\ Futu\ Strict\ r \implies$   
 $wf\_mbufn'\ \sigma\ P\ V\ j\ n\ R\ r\ buf \implies$   
 $wf\_ts\_regex\ \sigma\ P\ j\ r\ nts \implies$   
 $wf\_matchF\_aux\ \sigma\ V\ n\ R\ I\ r\ aux\ (progress\ \sigma\ P\ (Formula.MatchF\ I\ r)\ j)\ 0 \implies$   
 $progress\ \sigma\ P\ (Formula.MatchF\ I\ r)\ j + length\ aux = progress\_regex\ \sigma\ P\ r\ j \implies$   
 $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MMatchF\ I\ mr\ mrs\ \varphi s\ buf\ nts\ aux)\ (Formula.MatchF\ I\ r)$

**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*  $\sigma$   $V$   $R$  (*init\_args*  $I$   $n$  (*Formula.fv*  $\varphi'$ ) (*Formula.fv*  $\psi'$ )  $b$ )  $\varphi'$   $\psi'$  (*init\_muaux* (*init\_args*  $I$   $n$  (*Formula.fv*  $\varphi'$ ) (*Formula.fv*  $\psi'$ )  $b$ ))  $0$   
**unfolding** *wf\_until\_aux\_def* *wf\_until\_auxlist\_def* **by** (*auto intro: valid\_init\_muaux*)

**lemma** *wf\_matchP\_aux\_Nil*: *wf\_matchP\_aux*  $\sigma$   $V$   $n$   $R$   $I$   $r$  []  $0$   
**unfolding** *wf\_matchP\_aux\_def* **by** *simp*

**lemma** *wf\_matchF\_aux\_Nil*: *wf\_matchF\_aux*  $\sigma$   $V$   $n$   $R$   $I$   $r$  []  $0$   $k$   
**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*  $\varphi \implies \forall x \in \text{Formula.fv } \varphi. x < n \implies$   
*pred\_mapping*  $(\lambda x. x = 0)$   $P \implies$   
*wf\_mformula*  $\sigma$   $0$   $P$   $V$   $n$   $R$  (*init0*  $n$   $\varphi$ )  $\varphi$

**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*  $p$   $\varphi$   $\psi$ )

**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*  $\varphi$   $\psi$ )

**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*  $\varphi$   $\psi$ )

**then show** *?case* **by** (*auto intro!: wf\_mformula.And wf\_mbuf2'\_0*)

**next**

**case** (*And\_constraint*  $\varphi$   $\psi$ )

**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  $\varphi$  pnul 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  $\implies \forall t \in \text{set } ts. x \notin \text{Formula.fv\_trm } t \implies f x =$ 
  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  $\implies t \in \text{set } ts \implies x \in \text{Formula.fv\_trm } t \implies f x$ 
 $\neq$  None
  by (induction ts xs arbitrary: f rule: match.induct) (auto split: if_splits option.splits)

```

```

lemma match_eval_trm:  $\forall t \in \text{set } ts. \forall i \in \text{Formula.fv\_trm } t. i < n \implies$  Some f = match ts 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)  $\implies x \in A \implies x < n \implies \exists y. f x$ 
 $=$  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**  $\text{wf\_tuple } (b + n) \ (\text{Formula.fv } \varphi) \ (?zs \ @ \ v[y:=None])$   
**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**  $\neg \text{Formula.sat } \sigma \ V \ (\text{map the } (?zs \ @ \ v[y:=None])) \ i \ \varphi$   
**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**  $\text{Formula.sat } \sigma \ V \ (\text{map the } (?zs \ @ \ v[y:=None])) \ i \ \varphi \longleftrightarrow \text{Formula.sat } \sigma \ V \ (zs \ @ \ \text{map the } v) \ i \ \varphi$   
**using**  $\text{True}$  **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. let group = Set.filter ( $\lambda x. drop b x = k$ ) rel;
  images = meval_trm f ' group
  in ( $\lambda y. (y, ecard (Set.filter ( $\lambda x. meval_trm f x = y$ ) 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*  $\varphi s$ ) *mr*  $\implies$  *fv\_regex* (*from\_mregex* *mr*  $\varphi s$ )  $\subseteq$  ( $\bigcup \varphi \in \text{set } \varphi s. \text{fv } \varphi$ )  
**by** (*induct* *mr*) (*auto* *simp*: *Bex\_def* *in\_set\_conv\_nth*) $+$

**lemma** *qtable\_ε\_lax*:  
**assumes** *ok* (*length*  $\varphi s$ ) *mr*  
**and** *list\_all2* ( $\lambda \varphi \text{ rel. } \text{qtable } n \text{ (Formula.fv } \varphi) \text{ (mem_restr } R) (\lambda v. \text{Formula.sat } \sigma V \text{ (map the } v) i$   
 $\varphi) \text{ rel}) \varphi s \text{ rels}$   
**and** *fv\_regex* (*from\_mregex* *mr*  $\varphi s$ )  $\subseteq A$  **and** *qtable* *n* *A* (*mem\_restr* *R*) *Q* *guard*  
**shows** *qtable* *n* *A* (*mem\_restr* *R*)  
 $(\lambda v. \text{Regex.eps (Formula.sat } \sigma V \text{ (map the } v)) i \text{ (from_mregex } mr \varphi s) \wedge Q v) (\varepsilon\_lax \text{ guard rels } mr)$   
**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*  $\varphi s$ )  $\subseteq A$  *fv\_regex* (*from\_mregex* *mr2*  $\varphi s$ )  $\subseteq A$   
**using** *fv\_regex\_from\_mregex*[*of*  $\varphi s$  *mr1*] *fv\_regex\_from\_mregex*[*of*  $\varphi s$  *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*  $\implies$  ( $X = \text{empty\_table} \vee X = \text{unit\_table}$ )  
**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  $\varphi s$  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**  $\varphi s$  **where**

*ok\_lctxt*  $\varphi s$  *id id*  
 $| ok\_lctxt \varphi s \kappa \kappa' \implies ok\_lctxt \varphi s (\lambda t. \kappa (MTimes\ t\ mr)) (\lambda t. \kappa' (Regex.Times\ t\ (from\_mregex\ mr\ \varphi s)))$

**lemma** *ok\_lctxt\_swap*:  $ok\_lctxt \varphi s \kappa \kappa' \implies from\_mregex (\kappa\ mr) \varphi s = \kappa' (from\_mregex\ mr\ \varphi s)$   
**by** (*induct*  $\kappa \kappa'$  *arbitrary*: *mr* *rule*: *ok\_lctxt.induct*) *auto*

**lemma** *ok\_lctxt\_cong*:  $ok\_lctxt \varphi s \kappa \kappa' \implies Regex.match (Formula.sat\ \sigma\ V\ v)\ r = Regex.match (Formula.sat\ \sigma\ V\ v)\ s \implies$   
 $Regex.match (Formula.sat\ \sigma\ V\ v)\ (\kappa' r)\ i\ j = Regex.match (Formula.sat\ \sigma\ V\ v)\ (\kappa' s)\ i\ j$   
**by** (*induct*  $\kappa \kappa'$  *arbitrary*: *r s* *rule*: *ok\_lctxt.induct*) *simp\_all*

**lemma** *qtable\_ldk*:

**assumes**  $ok (length\ \varphi s)\ mr\ fv\_regex (from\_mregex\ mr\ \varphi s) \subseteq A$   
**and** *list\_all2*  $(\lambda \varphi\ rel.\ qtable\ n (Formula.fv\ \varphi) (mem\_restr\ R) (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ j\ \varphi)\ rel)\ \varphi s\ rels$   
**and** *ok\_lctxt*  $\varphi s \kappa \kappa'$   
**and**  $\forall ms \in \kappa' . LPD\ mr.\ qtable\ n\ A (mem\_restr\ R) (\lambda v.\ Q (map\ the\ v) (from\_mregex\ ms\ \varphi s)) (lookup\ rel\ ms)$   
**shows**  $qtable\ n\ A (mem\_restr\ R)$   
 $(\lambda v.\ \exists s \in Regex.lpd\ \kappa \kappa' (Formula.sat\ \sigma\ V\ (map\ the\ v))\ j (from\_mregex\ mr\ \varphi s).\ Q (map\ the\ v)\ s)$   
 $(ld\ \kappa\ rel\ rels\ mr)$   
**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**  $qtable\_ld = qtable\_ldk[OF\ _\ _\ _\ ok\_lctxt.intros(1),\ unfolded\ lpd\ \kappa\_lpd\ image\_id\ id\_apply]$

**lemma** *RPD\_fv\_regex\_le*:

$ms \in RPD\ mr \implies fv\_regex (from\_mregex\ ms\ \varphi s) \subseteq fv\_regex (from\_mregex\ mr\ \varphi s)$   
**by** (*induct* *mr* *arbitrary*: *ms*) (*auto simp: MTimesL\_def split: nat.splits*) $+$

**lemma** *RPD\_safe*:  $safe\_regex\ Past\ g (from\_mregex\ mr\ \varphi s) \implies$

$ms \in RPD\ mr \implies safe\_regex\ Past\ g (from\_mregex\ ms\ \varphi s)$

**proof** (*induct* *Past g from\_mregex mr  $\varphi s$  arbitrary: mr ms rule: safe\_regex\_induct*)

**case** *Skip*

**then show** *?case*

**by** (*cases* *mr*) (*auto split: nat.splits*)

**next**

**case** (*Test g  $\varphi$* )

```

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)  $\implies$ 
   $ms \in \text{RPDi } n \text{ mr} \implies \text{safe\_regex Past } g \text{ (from\_mregex } ms \varphi\text{s)}$ 
  by (induct n arbitrary: ms mr) (auto dest: RPD_safe)

lemma RPDs_safe: safe_regex Past g (from_mregex mr  $\varphi$ s)  $\implies$ 
   $ms \in \text{RPDs } mr \implies \text{safe\_regex Past } g \text{ (from\_mregex } ms \varphi\text{s)}$ 
  unfolding RPDs_def by (auto dest: RPDi_safe)

lemma RPD_safe_fv_regex: safe_regex Past Strict (from_mregex mr  $\varphi$ s)  $\implies$ 
   $ms \in \text{RPD } mr \implies \text{fv\_regex (from\_mregex } ms \varphi\text{s)} = \text{fv\_regex (from\_mregex } mr \varphi\text{s)}$ 
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  $\varphi$ s)  $\implies$ 
  ms  $\in$  RPDi 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: RPD_safe_fv_regex RPD_safe)

lemma RPDs_safe_fv_regex: safe_regex Past Strict (from_mregex mr  $\varphi$ s)  $\implies$ 
  ms  $\in$  RPDs mr  $\implies$  fv_regex (from_mregex ms  $\varphi$ s) = fv_regex (from_mregex mr  $\varphi$ s)
  unfolding RPDs_def by (auto dest: RPDi_safe_fv_regex)

lemma RPD_ok: ok m mr  $\implies$  ms  $\in$  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  $\in$  RPDi n mr  $\implies$  ok m ms
  by (induct n arbitrary: ms mr) (auto intro: RPD_ok)

lemma RPDs_ok: ok m mr  $\implies$  ms  $\in$  RPDs mr  $\implies$  ok m ms
  unfolding RPDs_def by (auto intro: RPDi_ok)

lemma LPD_fv_regex_le:
  ms  $\in$  LPD mr  $\implies$  fv_regex (from_mregex ms  $\varphi$ s)  $\subseteq$  fv_regex (from_mregex mr  $\varphi$ s)
  by (induct mr arbitrary: ms) (auto simp: MTimesR_def split: nat.splits)+

lemma LPD_safe: safe_regex Futu g (from_mregex mr  $\varphi$ s)  $\implies$ 
  ms  $\in$  LPD mr  $\implies$  safe_regex Futu g (from_mregex ms  $\varphi$ s)
proof (induct Futu g from_mregex mr  $\varphi$ s arbitrary: mr ms rule: safe_regex_induct)
  case Skip
  then show ?case
  by (cases mr) (auto split: nat.splits)
next
  case (Test g  $\varphi$ )
  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  $\varphi s$  arbitrary: mr rule: safe\_regex\_induct)

```

  case Skip
  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 (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

```

(Formula.MatchP (point ( $\tau \sigma ne - t$ )) (from\_mregex ms  $\varphi$ s))) (lookup X ms)) **for**  $t X$   
**unfolding** aux0\_def **using** safe mr mrs  
**by** (auto simp: lookup\_tabulate map\_of\_map\_restrict restrict\_map\_def finite\_RPDs \*\*\* RPDs\_trans  
 diff\_le\_mono2  
 intro!: sat\_MatchP\_rec[of  $\sigma \_ \_ ne$ , THEN iffD2]  
 qtable\_union[OF qtable\_rδ[OF  $\_ \_ qtables$ ] qtable\_rε\_strict[OF  $\_ \_ \_ qtables$ ],  
 of ms fv\_regex r λv r. Formula.sat  $\sigma V v (ne - Suc 0)$  (Formula.MatchP (point 0) r)  $\_ ms$  **for**  
 ms]  
 qtable\_cong[OF qtable\_rδ[OF  $\_ \_ qtables$ ],  
 of ms fv\_regex r λv r. Formula.sat  $\sigma V v (ne - Suc 0)$  (Formula.MatchP (point ( $\tau \sigma (ne - Suc$   
 0) -  $\tau \sigma i$ )) r)  
 $\_ \_ (\lambda v. Formula.sat \sigma V (map the v) ne (Formula.MatchP (point (\tau \sigma ne - \tau \sigma i)$   
 (from\_mregex ms  $\varphi$ s))) **for** ms  $i$ ]  
 dest!: assms(1)[unfolded wf\_matchP\_aux\_def, THEN conjunct2, THEN conjunct1, rule\_format]  
 sat\_MatchP\_rec[of  $\sigma \_ \_ ne$ , THEN iffD1]  
 elim!: bspec order.trans[OF  $\_ \_ \tau mono$ ] bexI[rotated] split: option.splits if\_splits)  
**then have** in\_aux0\_le\_τ:  $(t, X) \in set\ aux0 \implies t \leq \tau \sigma ne$  **for**  $t X$   
**by** (meson  $\tau mono$  diff\_le\_self le\_trans)  
**have** in\_aux0\_2:  $ne \neq 0 \implies t \leq \tau \sigma (ne-1) \implies \tau \sigma ne - t \leq right\ I \implies \exists i. \tau \sigma i = t \implies$   
 $\exists X. (t, X) \in set\ aux0$  **for**  $t$   
**proof** -  
**fix**  $t$   
**assume**  $ne \neq 0 t \leq \tau \sigma (ne-1) \tau \sigma ne - t \leq right\ I \exists i. \tau \sigma i = t$   
**then obtain**  $X$  **where**  $(t, X) \in set\ aux$   
**by** (atomize\_elim, intro assms(1)[unfolded wf\_matchP\_aux\_def, THEN conjunct2, THEN con-  
 junct2, rule\_format])  
 (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**  $(t, ?update\ X\ t) \in set\ aux0$   
**unfolding** aux0\_def **by** (auto simp: id\_def elim!: bexI[rotated] intro!: exI[of  $\_ X$ ])  
**then show**  $\exists X. (t, X) \in set\ aux0$   
**by** blast  
**qed**  
**have** aux0\_Nil:  $aux0 = [] \implies ne = 0 \vee ne \neq 0 \wedge (\forall t. t \leq \tau \sigma (ne-1) \wedge \tau \sigma ne - t \leq right\ I \implies$   
 $(\nexists i. \tau \sigma i = t))$   
**using** in\_aux0\_2 **by** (cases  $ne = 0$ ) (auto)  
  
**have** aux'\_eq:  $aux' = (case\ aux0\ of$   
 $[] \Rightarrow [(\tau \sigma ne, mrtabulate\ mrs\ (r\varepsilon\_strict\ n\ rels))]$   
 $| x \# aux' \Rightarrow (if\ fst\ x = \tau \sigma ne\ then\ x \# aux'$   
 $else\ (\tau \sigma ne, mrtabulate\ mrs\ (r\varepsilon\_strict\ n\ rels)) \# x \# aux')$   
**using** result\_eq **unfolding** aux0\_def update\_matchP\_def Let\_def **by** simp  
**have** sorted\_aux': sorted\_wrt  $(\lambda x y. fst\ x > fst\ y)$   $aux'$   
**unfolding** aux'\_eq  
**using** sorted\_aux0 in\_aux0\_le\_τ **by** (cases  $aux0$ ) (fastforce)+  
  
**have** in\_aux'\_1:  $t \leq \tau \sigma ne \wedge \tau \sigma ne - t \leq right\ I \wedge (\exists i. \tau \sigma i = t) \wedge$   
 $(\forall ms \in RPDs\ mr. qtable\ n (Formula.fv\_regex\ r) (mem\_restr\ R) (\lambda v.$   
 $Formula.sat\ \sigma\ V (map\ the\ v) ne (Formula.MatchP (point\ (\tau\ \sigma\ ne - t)) (from\_mregex\ ms\ \varphi s)))$   
 (lookup X ms))  
**if**  $aux'$ :  $(t, X) \in set\ aux'$  **for**  $t X$   
**proof** (cases  $aux0$ )  
**case** Nil  
**with**  $aux'$  **show** ?thesis  
**unfolding** aux'\_eq **using** safe mrs qtables aux0\_Nil \*  
**by** (auto simp: zero\_enat\_def[symmetric] sat\_MatchP\_rec[**where**  $i=ne$ ]  
 lookup\_tabulate finite\_RPDs split: option.splits  
 intro!: qtable\_cong[OF qtable\_rε\_strict]  
 dest: spec[of  $\_ \_ \tau \sigma (ne-1)$ ])

```

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) ( $\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) (ne + \text{length } \text{auxlist}) \varphi$ ) rel1
  and qtable2: qtable n (Formula.fv  $\psi$ ) (mem_restr R) ( $\lambda v. \text{Formula.sat } \sigma V (\text{map the } v) (ne + \text{length } \text{auxlist}) \psi$ ) 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 ( $\tau \sigma (ne + \text{length } \text{auxlist})$ )
  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*  $ms\ \varphi s$ ) *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*  
**{ 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  $\sigma$  V n R I r aux ne 0*  
**and** *safe: safe\_regex Futu Strict r*  
**and** *mr: to\_mregex r = (mr,  $\varphi$ s)*  
**and** *mrs: mrs = sorted\_list\_of\_set (LPDs mr)*  
**and** *nt: nt =  $\tau$   $\sigma$  (ne + length aux)*  
**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 + length aux)  $\varphi$ ) rel)  $\varphi$ s rels*  
**shows** *wf\_matchF\_aux  $\sigma$  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  $\sigma$  V n R pos  $\varphi$  I  $\psi$  (a # aux) ne  $\implies$*

*wf\_until\_auxlist  $\sigma$  V n R pos  $\varphi$  I  $\psi$  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  $\sigma$  V n R I r (entry # aux) ne i  $\implies$*

*wf\_matchF\_aux  $\sigma$  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  $\sigma$  V n R pos  $\varphi$  I  $\psi$  ((t, a1, a2) # aux) ne  $\implies$  t =  $\tau$   $\sigma$*   
*ne*

**unfolding** *wf\_until\_auxlist\_def*  
**by** (*simp add: upt\_conv\_Cons del: upt\_Suc*)

**lemma** *wf\_matchF\_aux\_Cons1: wf\_matchF\_aux  $\sigma$  V n R I r ((t, rels, rel) # aux) ne i  $\implies$  t =  $\tau$   $\sigma$*   
*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  $\sigma$  V n R pos  $\varphi$  I  $\psi$  ((t, a1, a2) # aux) ne  $\implies$*

*qtable n (Formula.fv  $\psi$ ) (mem\_restr R) ( $\lambda v$ . ( $\exists j$ . ne  $\leq$  j  $\wedge$  j < Suc (ne + length aux)  $\wedge$  mem ( $\tau$   $\sigma$  j -*  
 *$\tau$   $\sigma$  ne) I  $\wedge$*

*Formula.sat  $\sigma$  V (map the v) j  $\psi$   $\wedge$  ( $\forall k \in \{ne..<j\}$ . if pos then Formula.sat  $\sigma$  V (map the v) k  $\varphi$  else*  
 *$\neg$  Formula.sat  $\sigma$  V (map the v) k  $\varphi$ ))) a2*

**unfolding** *wf\_until\_auxlist\_def*  
**by** (*simp add: upt\_conv\_Cons del: upt\_Suc*)

**lemma** *wf\_matchF\_aux\_Cons3: wf\_matchF\_aux  $\sigma$  V n R I r ((t, rels, rel) # aux) ne i  $\implies$*

*qtable n (Formula.fv\_regex r) (mem\_restr R) ( $\lambda v$ . ( $\exists j$ . ne  $\leq$  j  $\wedge$  j < Suc (ne + length aux + i)  $\wedge$  mem*  
*( $\tau$   $\sigma$  j -  $\tau$   $\sigma$  ne) I  $\wedge$*

*Regex.match (Formula.sat  $\sigma$  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**  $ja \leq ja' \text{ } jb \leq jb'$   
**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**  $\text{length } Ps = \text{length } js' \text{ } \text{length } js' = \text{length } (\text{mbufn\_add } xss \text{ } buf)$   
**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**  
 $\text{length } [i..<js ! k] = \text{length } (buf ! k)$   
**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]*:  
 $\text{list\_all3 } P \text{ } xs \text{ } ys \text{ } [] \longleftrightarrow xs = [] \wedge ys = []$   
 $\text{list\_all3 } P \text{ } xs \text{ } [] \text{ } zs \longleftrightarrow xs = [] \wedge zs = []$   
 $\text{list\_all3 } P \text{ } [] \text{ } ys \text{ } zs \longleftrightarrow ys = [] \wedge zs = []$   
**unfolding** *list\_all3\_conv\_all\_nth* **by** *auto*

**lemma** *list\_all3\_Cons*:  
 $\text{list\_all3 } P \text{ } xs \text{ } ys \text{ } (z \# zs) \longleftrightarrow (\exists x \text{ } xs' \text{ } y \text{ } ys'. xs = x \# xs' \wedge ys = y \# ys' \wedge P x y z \wedge \text{list\_all3 } P \text{ } xs' \text{ } ys' \text{ } zs)$

$list\_all3\ P\ xs\ (y\ \# \ ys)\ zs \longleftrightarrow (\exists\ x\ xs'\ z\ zs'.\ xs = x\ \# \ xs' \wedge zs = z\ \# \ zs' \wedge P\ x\ y\ z \wedge list\_all3\ P\ xs'\ ys\ zs')$   
 $list\_all3\ P\ (x\ \# \ xs)\ ys\ zs \longleftrightarrow (\exists\ y\ ys'\ z\ zs'.\ ys = y\ \# \ ys' \wedge zs = z\ \# \ zs' \wedge P\ x\ y\ z \wedge list\_all3\ P\ xs\ ys'\ zs')$

**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 have ∀j∈set js. i ≤ j
by (auto simp: in_set_conv_nth list_all3_conv_all_nth)
moreover from True rec.prem2 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.prem3 nonempty True show ?thesis by simp
next
case False
from nonempty rec.prem2 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
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_Minis[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  $\bigwedge k xs z. i \leq k \implies \text{Suc } k \leq \text{Mini } i js \implies$ 
       $\text{list\_all2 } (\lambda P x. P k x) Ps xs \implies U k z \implies U (\text{Suc } k) (f xs z)$ 
  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 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** *mbuf2t\_take\_add'*:

**assumes**  $eq: mbuf2t\_take\ f\ z\ (mbuf2\_add\ xs\ ys\ buf)\ nts = (z', buf', nts')$   
**and** *bounded*:  $pred\_mapping\ (\lambda x. x \leq j)\ P\ pred\_mapping\ (\lambda x. x \leq j')\ P'$   
**and** *rm*:  $rel\_mapping\ (\leq)\ P\ P'$   
**and** *pre\_buf*:  $wf\_mbuf2'\ \sigma\ P\ V\ j\ n\ R\ \varphi\ \psi\ buf$   
**and** *pre\_nts*:  $list\_all2 (\lambda i\ t. t = \tau\ \sigma\ i)\ [min\ (progress\ \sigma\ P\ \varphi\ j)\ (progress\ \sigma\ P\ \psi\ j)..<j']\ nts$   
**and** *xs*:  $list\_all2 (\lambda i. qtable\ n\ (Formula.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi))$   
 $[progress\ \sigma\ P\ \varphi\ j..<progress\ \sigma\ P'\ \varphi\ j']\ xs$   
**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!*:  $mbuf2t\_take\_eqD[OF\ eq]\ wf\_mbuf2\_add[OF\ \_]\ xs\ ys]$   $progress\_mono\_gen[OF\ \langle j \leq j' \rangle$   
 $rm]$   
 $progress\_le\_gen)$

**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*:  $safe\_regex\ m\ g\ r \implies atms\ r = \{\} \longleftrightarrow regex.atms\ r = \{\}$

**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*:  $x \in (case\ y\ of\ formula.Neg\ \varphi' \Rightarrow \{\varphi'\} \mid \_ \Rightarrow \{\}) \longleftrightarrow y = formula.Neg\ x$

**by** (*cases y*) *auto*

**lemma** *atms\_nonempty\_progress*:

$safe\_regex\ m\ g\ r \implies atms\ r \neq \{\} \implies (\lambda \varphi. progress\ \sigma\ P\ \varphi\ j)\ 'atms\ r = (\lambda \varphi. progress\ \sigma\ P\ \varphi\ j)\ '$

*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,  $\varphi$ s)  $\implies$   $\varphi$ s  $\neq$  []  $\implies$  progress\_regex  $\sigma$  P r j = (MIN  $\varphi \in \text{set } \varphi$ s. progress  $\sigma$  P  $\varphi$  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,  $\varphi$ s)  $\implies$  progress\_regex  $\sigma$  P r j = (if  $\varphi$ s = [] then j else (MIN  $\varphi \in \text{set } \varphi$ s. progress  $\sigma$  P  $\varphi$  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,  $\varphi$ s)*  
**and** *pre\_buf: wf\_mbufn'  $\sigma$  P V j n R r buf*  
**and** *pre\_nts: list\_all2 ( $\lambda i t. t = \tau \sigma i$ ) [progress\_regex  $\sigma$  P r j.. $j'$ ] nts*  
**and** *xss: list\_all3 list\_all2*  
*(map ( $\lambda \varphi i. qtable n (fv \varphi) (mem_restr R) (\lambda v. Formula.sat \sigma V (map the v) i \varphi)) \varphi$ s)*  
*(map2 upt (map ( $\lambda \varphi. progress \sigma P \varphi j$ )  $\varphi$ s) (map ( $\lambda \varphi. progress \sigma P' \varphi j'$ )  $\varphi$ s)) xss*  
**and** *j  $\leq$  j'*  
**shows** *wf\_mbufn'  $\sigma$  P' V j' n R r buf'*  
**and** *wf\_ts\_regex  $\sigma$  P' j' r nts'*  
**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'  $\sigma$  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  $\varphi j$ .. $\langle$ progress  $\sigma$  P'  $\varphi j'$  $\rangle$ ] 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  $\sigma$  P  $\psi j$ .. $\langle$ progress  $\sigma$  P'  $\psi j'$  $\rangle$ ] ys*  
**and** *j  $\leq$  j'*  
**and** *base: U (min (progress  $\sigma$  P  $\varphi j$ ) (progress  $\sigma$  P  $\psi j$ )) 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  $\sigma$  P'  $\varphi j' \implies$  Suc k  $\leq$  progress  $\sigma$  P'  $\psi j' \implies$*   
*qtable n (Formula.fv  $\varphi$ ) (mem\_restr R) ( $\lambda v. Formula.sat \sigma V (map the v) k \varphi$ ) X  $\implies$*   
*qtable n (Formula.fv  $\psi$ ) (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  $\sigma$  P'  $\varphi j'$ ) (progress  $\sigma$  P'  $\psi j')) z'$*   
**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,  $\varphi$ s)*  
**and** *pre\_buf*: *wf\_mbufn'*  $\sigma P V j n R r$  *buf*  
**and** *pre\_nts*: *list\_all2* ( $\lambda i t. t = \tau \sigma i$ ) [*progress\_regex*  $\sigma P r j..<j'$ ] *nts*  
**and** *xss*: *list\_all3 list\_all2*  
(*map* ( $\lambda \varphi i. qtable\ n\ (fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi))\ \varphi s$ )  
(*map2* *upt* (*map* ( $\lambda \varphi. progress\ \sigma\ P\ \varphi\ j$ )  $\varphi s$ ) (*map* ( $\lambda \varphi. progress\ \sigma\ P'\ \varphi\ j'$ )  $\varphi s$ )) *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* ( $\lambda \varphi. qtable\ n\ (Formula.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v. Formula.sat\ \sigma\ V\ (map\ the\ v)\ k\ \varphi))\ \varphi s$   
*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*  $\circ$  *g*) *xs*  
**by** (*induct* *xs*) *auto*

**lemma** *map\_split\_alt*: *map\_split* *f* *xs* = (*map* (*fst*  $\circ$  *f*) *xs*, *map* (*snd*  $\circ$  *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*  $\sigma$  *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*  $\sigma \ j \ P \ P' \ V \ db =$   
 $(\text{dom } V = \text{dom } P \wedge$   
 $\text{Mapping.keys } db = \text{dom } P \cup \{p. p \in \text{fst } \Gamma \ \sigma \ j\} \wedge$   
 $\text{rel\_mapping } (\leq) \ P \ P' \wedge$   
 $\text{pred\_mapping } (\lambda i. i \leq j) \ P \wedge$   
 $\text{pred\_mapping } (\lambda i. i \leq \text{Suc } j) \ P' \wedge$   
 $(\forall p \in \text{Mapping.keys } db - \text{dom } P. \text{the } (\text{Mapping.lookup } db \ p) = [\{ts. (p, ts) \in \Gamma \ \sigma \ j\}] \wedge$   
 $(\forall p \in \text{dom } P. \text{list\_all2 } (\lambda i \ X. X = \text{the } (V \ p) \ i) [\text{the } (P \ p)..<\text{the } (P' \ p)] \ (\text{the } (\text{Mapping.lookup } db \ p))))$

**lift\_definition** *mk\_db* ::  $(\text{Formula.name} \times \text{event\_data list}) \ \text{set} \Rightarrow \text{Formula.database is}$   
 $\lambda X \ p. (\text{if } p \in \text{fst } \Gamma \ X \ \text{then } \text{Some } [\{ts. (p, ts) \in X\}] \ \text{else } \text{None}) .$

**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*  $(\lambda i. \text{qtable } m \ (\text{Formula.fv } \varphi) \ (\text{mem\_restr } \text{UNIV}) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map the } v) \ i \ \varphi))$   
 $[\text{progress } \sigma \ P \ \varphi \ j..<\text{progress } \sigma \ P' \ \varphi \ (\text{Suc } j)] \ xs$   
**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*  $(\lambda x \ y. \text{map the } \Gamma \ y = \{v. \text{length } v = m \wedge \text{Formula.sat } \sigma \ V \ v \ x \ \varphi\})$   
 $[\text{progress } \sigma \ P \ \varphi \ j..<\text{progress } \sigma \ P' \ \varphi \ (\text{Suc } j)] \ xs$   
**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*  $(\lambda i \ X. X = \text{the } (V \ p') \ i) [\text{the } (P \ p')..<\text{the } (P' \ p')]$  (*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_regex_le[simp]:
  wf_envs σ j P P' V db ⇒ progress_regex σ P r j ≤ j
  wf_envs σ j P P' V db ⇒ progress_regex σ P' r (Suc j) ≤ Suc j
unfolding wf_envs_def by (auto simp: progress_regex_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  $\subseteq$  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 ⋃ (fv ' set l') = ⋃ (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 (⋃ (fv ' 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) ↔
¬ 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) ∧
  list_all2 (λA Qi. ¬ 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 ⇒

```

```

  ¬ 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..by auto
  with Prev(4,5) MPrev.premis(2) have list_all2 (λi X. if mem (τ σ (Suc i) - τ σ i) I then ?P i X
  else X = empty_table)
  [min (progress σ P ψ j) (j - 1)..by (intro mprev) (auto intro!: list_all2_upt_append list_all2_appendI order.trans[OF min.cobounded1])
  moreover have min (Suc (progress σ P ψ j)) j = Suc (min (progress σ P ψ j) (j-1)) if j > 0
  using that by auto
  ultimately show ?thesis using Prev(1,3) MPrev.premis(2) IH
  by (auto simp: map_Suc_upt[symmetric] upt_Suc[of 0] list.rel_map qtable_empty_iff
   simp del: upt_Suc elim!: wf_mformula.Prev list.rel_mono_strong
   split: prod.split if_split_asm)
qed
next
case (MNext I φ first nts)
from MNext.premis show ?case
proof (cases rule: wf_mformula.cases)
  case (Next ψ)

  have min[simp]:

```

```

min (progress  $\sigma$   $P$   $\psi$   $j - \text{Suc } 0$ ) ( $j - \text{Suc } 0$ ) = progress  $\sigma$   $P$   $\psi$   $j - \text{Suc } 0$ 
min (progress  $\sigma$   $P'$   $\psi$  ( $\text{Suc } j - \text{Suc } 0$ ))  $j$  = progress  $\sigma$   $P'$   $\psi$  ( $\text{Suc } j - \text{Suc } 0$ )
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  $\sigma$   $P'$   $\psi$  ( $\text{Suc } j - 1$ )) ( $\text{Suc } j - 1$ )
from Next MNext.IH[OF _ MNext.premis(2), of n R  $\psi$ ] have IH: wf_mformula  $\sigma$  ( $\text{Suc } j$ )  $P' V n R$ 
? $\varphi$   $\psi$ 
  list_all2 ?P [progress  $\sigma$   $P$   $\psi$   $j$ .. $\text{progress } \sigma$   $P'$   $\psi$  ( $\text{Suc } j$ )] ?xs by auto
  with Next have list_all2 ( $\lambda i$  X. if mem ( $\tau$   $\sigma$  ( $\text{Suc } i$ ) -  $\tau$   $\sigma$   $i$ ) I then ?P ( $\text{Suc } i$ ) X else X =
empty_table)
    [progress  $\sigma$   $P$   $\psi$   $j - 1$ .. $\text{?min}$ ] ?ls  $\wedge$ 
    list_all2 ?P [ $\text{Suc } \text{?min}$ .. $\text{progress } \sigma$   $P'$   $\psi$  ( $\text{Suc } j$ )] ?rs  $\wedge$ 
    list_all2 ( $\lambda i$  t. t =  $\tau$   $\sigma$   $i$ ) [ $\text{?min}$ .. $\text{Suc } j$ ] ?ts if progress  $\sigma$   $P$   $\psi$   $j$  < progress  $\sigma$   $P'$   $\psi$  ( $\text{Suc } j$ )
  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  $j$   $\text{Suc } j$   $\psi$ , simplified] Next IH
  by (cases progress  $\sigma$   $P'$   $\psi$  ( $\text{Suc } j$ ) > 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:  $\varphi' = \text{Formula.Since } \varphi''' I \psi''$ 
  and pos: if args_pos args then  $\varphi''' = \varphi''$  else  $\varphi''' = \text{Formula.Neg } \varphi''$ 
  and pos_eq: safe_formula  $\varphi''' = \text{args\_pos args}$ 
  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  $\varphi'' \subseteq \text{Formula.fv } \psi''$ 
  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  $\varphi''' = \text{Formula.fv } \varphi''$  progress  $\sigma$   $P$   $\varphi''' j = \text{progress } \sigma$   $P$   $\varphi'' j$ 
  progress  $\sigma$   $P'$   $\varphi''' j = \text{progress } \sigma$   $P'$   $\varphi'' j$  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  $\sigma$   $P$   $\psi'' j$ ).. $\text{Suc } j$ ] (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  $\varphi''' I$ 
 $\psi''$ ) ( $\text{Suc } j$ ))  $\wedge$ 
  list_all2 ( $\lambda i$ . qtable n (Formula.fv  $\varphi''' \cup \text{Formula.fv } \psi''$ ) (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 []  $\Rightarrow$   $\tau \sigma j$  | nt # _  $\Rightarrow$  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  $\leq$  progress  $\sigma P'$  (Formula.Until  $\varphi''' I \psi''$ ) (Suc j)  $\Longrightarrow$ 
wf_until_auxlist  $\sigma V n R$  pos  $\varphi'' I \psi''$  auxlist' i  $\Longrightarrow$ 
i + length auxlist' = min (progress  $\sigma P' \varphi'''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j))  $\Longrightarrow$ 
wf_until_auxlist  $\sigma V n R$  pos  $\varphi'' I \psi''$  auxlist'' (progress  $\sigma P'$  (Formula.Until  $\varphi''' I \psi''$ ) (Suc j))  $\wedge$ 
i + length zs = progress  $\sigma P'$  (Formula.Until  $\varphi''' I \psi''$ ) (Suc j)  $\wedge$ 
i + length zs + length auxlist'' = min (progress  $\sigma P' \varphi'''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j))  $\wedge$ 
list_all2 ( $\lambda i. qtable n$  (Formula.fv  $\psi''$ ) (mem_restr R)
( $\lambda v. Formula.sat \sigma V$  (map the v) i (Formula.Until (if pos then  $\varphi''$  else Formula.Neg  $\varphi''$ ) I  $\psi''$ )))
[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  $\sigma V n R$  pos  $\varphi'' I \psi''$  aux' (Suc i)
by (rule wf_until_aux_Cons)
from Cons.prem1(2) have 1: t =  $\tau \sigma i$ 
unfolding  $\langle a = (t, a1, a2) \rangle$  by (rule wf_until_aux_Cons1)
from Cons.prem1(2) have 3: qtable n (Formula.fv  $\psi''$ ) (mem_restr R) ( $\lambda v. (\exists j \geq i. j < Suc (i + length aux') \wedge mem (\tau \sigma j - \tau \sigma i) I \wedge Formula.sat \sigma V$  (map the v) j  $\psi'' \wedge (\forall k \in \{i..<j\}. if pos then Formula.sat \sigma V$  (map the v) k  $\varphi''$  else  $\neg Formula.sat \sigma V$  (map the v) k  $\varphi''$ ))) a2
unfolding  $\langle a = (t, a1, a2) \rangle$  by (rule wf_until_aux_Cons3)
from Cons.prem1(3) have Suc_i_aux': Suc i + length aux' =
min (progress  $\sigma P' \varphi'''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j))
by simp
have i  $\geq$  progress  $\sigma P'$  (Formula.Until  $\varphi''' I \psi''$ ) (Suc j)
if enat (case nts' of []  $\Rightarrow$   $\tau \sigma j$  | nt # x  $\Rightarrow$  nt)  $\leq$  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  $\varphi'''$ 
intro!: cInf_lower  $\tau$ _mono elim!: order.trans[rotated] simp del: upt_Suc split: if_splits list.splits)
moreover
have Suc i  $\leq$  progress  $\sigma P'$  (Formula.Until  $\varphi''' I \psi''$ ) (Suc j)
if enat t + right I < enat (case nts' of []  $\Rightarrow$   $\tau \sigma j$  | nt # x  $\Rightarrow$  nt)
proof -
from that obtain m where m: right I = enat m by (cases right I) auto
have  $\tau$ _min:  $\tau \sigma$  (min j k) = min ( $\tau \sigma j$ ) ( $\tau \sigma k$ ) for k
by (simp add: min_of_mono monoI)
have le_progress_iff[simp]: (Suc j)  $\leq$  progress  $\sigma P' \varphi$  (Suc j)  $\longleftrightarrow$  progress  $\sigma P' \varphi$  (Suc j) = (Suc j) for  $\varphi$ 
using wf_envs_progress_le[OF MUntil.prem1(2), of  $\varphi$ ] by auto
have min_Suc[simp]: min j (Suc j) = j by auto
let ?X = {i.  $\forall k. k < Suc j \wedge k \leq$  min (progress  $\sigma P' \varphi'''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j))  $\longrightarrow$ 
enat ( $\tau \sigma k$ )  $\leq$  enat ( $\tau \sigma i$ ) + right I}
let ?min = min j (min (progress  $\sigma P' \varphi''$  (Suc j)) (progress  $\sigma P' \psi''$  (Suc j)))
have  $\tau \sigma$  ?min  $\leq$   $\tau \sigma j$ 
by (rule  $\tau$ _mono) auto
from m have ?X  $\neq$  {}
by (auto dest!:  $\tau$ _mono[of _ progress  $\sigma P' \varphi''$  (Suc j)  $\sigma$ ])

```

```

      simp: not_le not_less  $\varphi'''$  intro!: exI[of _ progress  $\sigma$   $P'$   $\varphi''$  (Suc j)]
from m show ?thesis
  using that nts' unfolding wf_ts_def progress.simps
  by (intro cInf_greatest[OF  $\langle ?X \neq \{\} \rangle$ ])
      (auto simp: 1  $\varphi'''$  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  $\tau \sigma i + m \tau \sigma \_ \tau \sigma \_ + m$ ] less_τD)
qed
moreover have *:  $k < \text{progress } \sigma \ P' \ \psi \ (\text{Suc } j)$  if
  enat ( $\tau \sigma i$ ) + right  $I < \text{enat } (\text{case } nts' \text{ of } [] \Rightarrow \tau \sigma j \mid nt \# x \Rightarrow nt)$ 
  enat ( $\tau \sigma k - \tau \sigma i$ )  $\leq$  right  $I \ \psi = \psi'' \vee \psi = \varphi''$  for  $k \ \psi$ 
proof -
  from that(1,2) obtain m where right  $I = \text{enat } m$ 
   $\tau \sigma i + m < (\text{case } nts' \text{ of } [] \Rightarrow \tau \sigma j \mid nt \# x \Rightarrow nt) \tau \sigma k - \tau \sigma i \leq m$ 
  by (cases right  $I$ ) auto
  with that(3) nts' progress_le[of  $\sigma \ \psi''$  Suc j] progress_le[of  $\sigma \ \varphi''$  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  $\tau \sigma k$ ] less_τD)
qed
ultimately show ?case using Cons.premS Suc_i_aux'[simplified]
  unfolding  $\langle a = (t, a1, a2) \rangle$ 
  by (auto simp:  $\varphi'''$  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.premS(2) le_SucI[OF order_refl]]
  wf_auxlist' conjunct2[OF update1, unfolded ne_def length_aux']]
have progress  $\sigma \ P$  (formula.Until  $\varphi'''$   $I \ \psi''$ )  $j + \text{length } \text{auxlist}' =$ 
  progress  $\sigma \ P'$  (formula.Until  $\varphi'''$   $I \ \psi''$ ) (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  $\sigma \ P'$  (formula.Until  $\varphi'''$   $I \ \psi''$ ) (Suc j) + length auxlist'' = 0 then 0
  else  $\tau \sigma$  (progress  $\sigma \ P'$  (formula.Until  $\varphi'''$   $I \ \psi''$ ) (Suc j) + length auxlist'' - 1))
unfolding cur ne_def by auto
then have wf_until_aux  $\sigma \ V \ R$  args  $\varphi'' \ \psi'' \ \text{aux}''$  (progress  $\sigma \ P'$  (formula.Until  $\varphi'''$   $I \ \psi''$ ) (Suc j))  $\wedge$ 
  progress  $\sigma \ P$  (formula.Until  $\varphi'''$   $I \ \psi''$ )  $j + \text{length } \text{zs} = \text{progress } \sigma \ P'$  (formula.Until  $\varphi'''$   $I \ \psi''$ ) (Suc
j)  $\wedge$ 
  progress  $\sigma \ P$  (formula.Until  $\varphi'''$   $I \ \psi''$ )  $j + \text{length } \text{zs} + \text{length\_muaux } \text{args } \text{aux}'' = \min$  (progress  $\sigma$ 
 $P'$   $\varphi'''$  (Suc j)) (progress  $\sigma \ P'$   $\psi''$  (Suc j))  $\wedge$ 
  list_all2 ( $\lambda i. \text{qtable } n \ (\text{fv } \psi'') \ (\text{mem\_restr } R) \ (\lambda v. \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i) \ (\text{formula.Until}$ 
(if pos then  $\varphi''$  else formula.Neg  $\varphi''$ )  $I \ \psi''$ ))
  [progress  $\sigma \ P$  (formula.Until  $\varphi'''$   $I \ \psi''$ )  $j..<$ progress  $\sigma \ P$  (formula.Until  $\varphi'''$   $I \ \psi''$ )  $j + \text{length } \text{zs}$ ]  $\text{zs}$ 
  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  $\varphi$  MUntil.premS(2)] MUntil.IH(2)[OF  $\psi$  MUntil.premS(2)] pos pos_eq fvi_subset
show ?case
  by (auto 0 4 simp: args_ivl args_n args_pos Until_eq  $\varphi'''$  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.premS(2)] buf nts_snoc]
      mbuf2t_take_add'(2)[OF _ wf_envs_P_simps[OF MUntil.premS(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 ( $\lambda i. \ qtable \ n \ (\text{Formula.fv\_regex } r) \ (\text{mem\_restr } R)$ 
  ( $\lambda v. \ \text{Formula.sat } \sigma \ V \ (\text{map } \text{the } v) \ i \ (\text{Formula.MatchP } I \ r)$ ))
  [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.premis(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.premis(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.premis(2)]
elim: wf_update_matchF[OF safe mr mrs] elim!: list.rel_mono_strong)
from MMatchF.premis(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.premis(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.premis(2) have aux': wf_matchF_aux  $\sigma$  V n R I r aux' (Suc i) 0
by (rule wf_matchF_aux_Cons)
from Cons.premis(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.prem(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.prem(2)]]
safe mr buf_nts_snoc]
      mbufnt_take_add'(2)[OF _ wf_envs_P_simps[OF MMatchF.prem(2)]] safe mr buf_nts_snoc]
      dest!: MMatchF.IH[OF _ MMatchF.prem(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  $\sigma$  Map.empty  $\varphi$  (plen  $\pi$ ))])  
**ultimately show** ?thesis  
**using**  $assms(4)$  **unfolding** *mstep\_def* *Let\_def* *flatten\_verdicts\_def*  
**by** (auto simp: in\_set\_indexed\_from\_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  $\sigma$  Map.empty  $\varphi$  (plen  $\pi$ )))]))

**qed**

#### 6.6.5 Monitor function

**locale** *verimon* = *verimon\_spec* + *maux*

**lemma** (in *verimon*) *mstep\_mverdicts*:

**assumes** *wf*: *wf\_mstate*  $\varphi$   $\pi$  *R* *st*

**and** *le[simp]*: *last\_ts*  $\pi \leq \text{snd } \text{tdb}$

**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**  $\sigma$  **where** *p2*: *prefix\_of* (*psnoc*  $\pi$  *tdb*)  $\sigma$

**using** *ex\_prefix\_of* **by** *blast*

**with** *le* **have** *p1*: *prefix\_of*  $\pi$   $\sigma$  **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* \_  $\sigma$ ]

*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* #  $\pi$ ) *st* =

(*let* (*V'*, *st'*) = *mstep* (*map\_prod* *mk\_db* *id* *tdb*) *st*; (*V''*, *st''*) = *msteps0*  $\pi$  *st'* *in* (*V'* @ *V''*, *st''*))

**primrec** *msteps0\_stateless* **where**

*msteps0\_stateless* [] *st* = []

| *msteps0\_stateless* (*tdb* #  $\pi$ ) *st* = (*let* (*V'*, *st'*) = *mstep* (*map\_prod* *mk\_db* *id* *tdb*) *st* *in* *V'* @ *msteps0\_stateless*  $\pi$  *st'*)

**lemma** *msteps0\_msteps0\_stateless*: *fst* (*msteps0* *w* *st*) = *msteps0\_stateless* *w* *st*

**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* ( $\pi$  @ [*tdb*]) *st* =

(*let* (*V'*, *st'*) = *msteps0*  $\pi$  *st*; (*V''*, *st''*) = *mstep* (*map\_prod* *mk\_db* *id* *tdb*) *st'* *in* (*V'* @ *V''*, *st''*))

**by** (*induct*  $\pi$  *arbitrary*: *st*) (*auto split*: *prod\_splits*)

**lemma** *msteps\_psnoc*: *last\_ts*  $\pi \leq \text{snd } \text{tdb} \implies$  *msteps* (*psnoc*  $\pi$  *tdb*) *st* =

(*let* (*V'*, *st'*) = *msteps*  $\pi$  *st*; (*V''*, *st''*) = *mstep* (*map\_prod* *mk\_db* *id* *tdb*) *st'* *in* (*V'* @ *V''*, *st''*))

**by** *transfer'* (*auto simp*: *msteps0\_snoc split*: *list\_splits prod\_splits if\_splits*)

**definition** *monitor* **where**

*monitor*  $\varphi$   $\pi$  = *msteps\_stateless*  $\pi$  (*minit\_safe*  $\varphi$ )

**end**

**lemma** *Suc\_length\_conv\_snoc*:  $(\text{Suc } n = \text{length } xs) = (\exists y \text{ ys}. xs = ys @ [y] \wedge \text{length } ys = n)$   
**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*)<sup>+</sup>

**with** 0(2) **show** ?*case unfolding flatten\_verdicts\_def by simp*

**next**

**case** (*Suc x*)

**from** *Suc(2,5)* **obtain**  $\pi''$  *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*)<sup>+</sup>

**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** *init\_safe\_init*:  $mmonitorable \varphi \implies init\_safe \varphi = init \varphi$

**unfolding** *init\_safe\_def monitorable\_formula\_code* **by** *simp*

**lemma** *wf\_mstate\_init\_safe*:  $mmonitorable \varphi \implies wf\_mstate \varphi pnil R (init\_safe \varphi)$

**using**  $wf\_mstate\_init init\_safe\_init mmonitorable\_def$  **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 # *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 queue\_t  
**assume** queue\_invariant q

**then show**  $\text{pred\_prod} \top \text{queue\_invariant} (\text{safe\_hd\_t } q)$   
**by** ( $\text{cases } q \text{ rule: safe\_hd\_t.cases}$ ) ( $\text{auto simp: queue\_invariant\_def Let\_def split: list.split}$ )  
**qed**

**lemma**  $\text{safe\_hd\_rep: safe\_hd } q = (\alpha, q') \implies \text{linearize } q = \text{linearize } q' \wedge$   
 $(\text{case } \alpha \text{ of None} \implies \text{linearize } q = [] \mid \text{Some } a \implies \text{linearize } q \neq [] \wedge a = \text{hd} (\text{linearize } q))$   
**by** *transfer*  
 $(\text{auto simp add: queue\_invariant\_def Let\_def hd\_append split: list.splits elim: safe\_hd\_t.elims})$

**fun**  $\text{replace\_hd\_t} :: 'a \Rightarrow 'a \text{ queue\_t} \Rightarrow 'a \text{ queue\_t}$  **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**  $\text{replace\_hd} :: 'a \Rightarrow 'a \text{ queue} \Rightarrow 'a \text{ queue}$  **is**  $\text{replace\_hd\_t}$   
**by** ( $\text{auto simp: queue\_invariant\_def split: list.splits elim: replace\_hd\_t.elims}$ )

**lemma**  $\text{tl\_append: } xs \neq [] \implies \text{tl } xs @ ys = \text{tl} (xs @ ys)$   
**by** *simp*

**lemma**  $\text{replace\_hd\_rep: linearize } q = f \# fs \implies \text{linearize} (\text{replace\_hd } a q) = a \# fs$   
**proof** (*transfer fixing: f fs a*)

**fix**  $q$   
**assume**  $\text{queue\_invariant } q$  **and**  $(\text{case } q \text{ of } (fs, ls) \Rightarrow fs @ \text{rev } ls) = f \# fs$   
**then show**  $(\text{case } \text{replace\_hd\_t } a q \text{ of } (fs, ls) \Rightarrow fs @ \text{rev } ls) = a \# fs$   
**by** ( $\text{cases } (a, q) \text{ rule: replace\_hd\_t.cases}$ ) ( $\text{auto simp: queue\_invariant\_def tl\_append}$ )  
**qed**

**fun**  $\text{replace\_last\_t} :: 'a \Rightarrow 'a \text{ queue\_t} \Rightarrow 'a \text{ queue\_t}$  **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**  $\text{replace\_last} :: 'a \Rightarrow 'a \text{ queue} \Rightarrow 'a \text{ queue}$  **is**  $\text{replace\_last\_t}$   
**by** ( $\text{auto simp: queue\_invariant\_def split: list.splits elim: replace\_last\_t.elims}$ )

**lemma**  $\text{replace\_last\_rep: linearize } q = fs @ [f] \implies \text{linearize} (\text{replace\_last } a q) = fs @ [a]$   
**by** *transfer* ( $\text{auto simp: queue\_invariant\_def split: list.splits prod.splits elim!: replace\_last\_t.elims}$ )

**fun**  $\text{tl\_queue\_t} :: 'a \text{ queue\_t} \Rightarrow 'a \text{ queue\_t}$  **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**  $\text{tl\_queue} :: 'a \text{ queue} \Rightarrow 'a \text{ queue}$  **is**  $\text{tl\_queue\_t}$   
**by** ( $\text{auto simp: queue\_invariant\_def split: list.splits elim!: tl\_queue\_t.elims}$ )

**lemma**  $\text{tl\_queue\_rep: } \neg \text{is\_empty } q \implies \text{linearize} (\text{tl\_queue } q) = \text{tl} (\text{linearize } q)$   
**by** *transfer* ( $\text{auto simp: tl\_append split: prod.splits list.splits elim!: tl\_queue\_t.elims}$ )

**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* ( $\text{auto split: prod.splits list.splits elim: tl\_queue\_t.elims}$ )

**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)  $\implies \forall t \in \text{fst } ' \text{ set } xs. t \leq nt \implies$

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)  $\implies \forall t \in \text{fst } ' \text{ set } xs. t \leq nt \implies$

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)  $\implies \forall t \in \text{fst } ' \text{ set } xs. t \leq nt \implies$

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 (set (linearize data_in')). valid_tuple tuple_since tas  $\wedge$  as = 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  $\in X$ 
        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**  $Cons.premis$  **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* '  $\{(t, X). (t, X) \in \text{set } (x \# xs) \wedge as \in Z X t\}$ ) =  
 $((\text{fst ' } \{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\}) \cup$   
 $(\text{if } as \in Z X t \text{ then } \{t\} \text{ else } \{\}))$   
**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* '  $\{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\}$ )) =  
 $\text{Max } (\text{fst ' } \{(t, X). (t, X) \in \text{set } xs \wedge as \in Z X t\})$   
**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*) =  
 $(\text{let } I = \text{args\_ivol } args;$   
 $(\text{data\_prev, move}) = \text{takedownWhile\_queue } (\lambda(t, X). nt - t \geq \text{left } I) \text{ data\_prev};$   
 $\text{data\_in} = \text{fold } (\lambda(t, X) \text{ data\_in. } \text{append\_queue } (t, X) \text{ data\_in}) \text{ move } \text{data\_in};$   
 $\text{tuple\_in} = \text{fold } (\lambda(t, X) \text{ tuple\_in. } \text{upd\_set } \text{tuple\_in } (\lambda_. t)$   
 $\{\text{as} \in X. \text{valid\_tuple } \text{tuple\_since } (t, \text{as})\}) \text{ move } \text{tuple\_in } \text{in}$   
 $(nt, gc, \text{maskL}, \text{maskR}, \text{data\_prev}, \text{data\_in}, \text{tuple\_in}, \text{tuple\_since}))$

**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*  
 $(ot, gc, \text{maskL}, \text{maskR}, \text{data\_prev}, \text{data\_in}, \text{tuple\_in}, \text{tuple\_since}) \text{ auxlist}$   
**and** *nt\_mono*:  $nt \geq cur$   
**shows** *valid\_mmsaux args nt* (*add\_new\_ts\_mmsaux'* *args nt*  
 $(ot, gc, \text{maskL}, \text{maskR}, \text{data\_prev}, \text{data\_in}, \text{tuple\_in}, \text{tuple\_since})) \text{ auxlist}$

**proof** –  
**define** *I* **where**  $I = \text{args\_ivol } args$

```

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**  $\text{join\_mmsaux\_abs } X (nt, gc, maskL, maskR, data\_prev, data\_in, tuple\_in, tuple\_since) =$   
 $\text{join\_mmsaux\_abs } X (nt, gc, maskL, maskR, data\_prev, data\_in, tuple\_in, tuple\_since)$

**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*:  $\bigwedge as. as \in \text{Mapping.keys } tuple\_in \implies wf\_tuple\ n\ L\ as$

**using** *wf\_tuple\_change\_base valid\_before True* **by** (*fastforce simp add: table\_def n\_def L\_def*)

**have** *cong\_in*:  $\bigwedge as\ n. as \in \text{Mapping.keys } tuple\_in \implies$

*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*:  $\bigwedge as. as \in \text{Mapping.keys } tuple\_since \implies wf\_tuple\ n\ L\ as$

**using** *wf\_tuple\_change\_base valid\_before True* **by** (*fastforce simp add: table\_def n\_def L\_def*)

**have** *cong\_since*:  $\bigwedge as\ n. as \in \text{Mapping.keys } tuple\_since \implies$

*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*  $\forall i \in \text{set } maskL. \neg i$ )

**case** *True*

**have** *length\_maskL*: *length maskL* = *n*

**using** *valid\_before* **by** (*auto simp add: join\_mask\_def n\_def*)

**have** *proj\_rep*:  $\bigwedge as. wf\_tuple\ n\ R\ as \implies proj\_tuple\ maskL\ as = replicate\ (length\ maskL)\ None$

**using** *True proj\_tuple\_replicate* **by** (*force simp add: length\_maskL wf\_tuple\_def*)

**have** *keys\_wf\_in*:  $\bigwedge as. as \in \text{Mapping.keys } tuple\_in \implies wf\_tuple\ n\ R\ as$

```

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  $\implies$ 
  table (args_n args) (args_L args) X  $\implies$  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  $\implies$ 
  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)
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))  $\implies$  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))  $\implies$ 
  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*:  $(a, b) \in X \implies b \in \text{snd } 'X$

**by** *force*

**fun** *result\_mmsaux* ::  $\text{args} \Rightarrow 'a \text{ mmsaux} \Rightarrow 'a \text{ table}$  **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*:  $\text{valid\_mmsaux } \text{args } \text{cur } \text{aux } \text{auxlist} \implies$

```

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**  $'a \text{ mmuaux}$  =  $\text{tp} \times \text{ts } \text{queue} \times \text{nat} \times \text{bool } \text{list} \times \text{bool } \text{list} \times$

```

  ('a tuple, tp) mapping × (tp, ('a tuple, ts + tp) mapping) mapping × 'a table list × nat

```

**definition** *tstp\_lt* ::  $\text{ts} + \text{tp} \Rightarrow \text{ts} \Rightarrow \text{tp} \Rightarrow \text{bool}$  **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*  $\longleftrightarrow$   
 $\text{args\_L } \text{args} \subseteq \text{args\_R } \text{args} \wedge$   
 $\text{maskL} = \text{join\_mask } (\text{args\_n } \text{args}) (\text{args\_L } \text{args}) \wedge$   
 $\text{maskR} = \text{join\_mask } (\text{args\_n } \text{args}) (\text{args\_R } \text{args}) \wedge$   
 $\text{len} \leq tp \wedge$   
 $\text{length } (\text{linearize } tss) = \text{len} \wedge \text{sorted } (\text{linearize } tss) \wedge$   
 $(\forall t \in \text{set } (\text{linearize } tss). t \leq \text{cur} \wedge \text{enat } \text{cur} \leq \text{enat } t + \text{right } (\text{args\_ivl } \text{args})) \wedge$   
 $\text{table } (\text{args\_n } \text{args}) (\text{args\_L } \text{args}) (\text{Mapping.keys } a1\_map) \wedge$   
 $\text{Mapping.keys } a2\_map = \{tp - \text{len}..tp\} \wedge$   
 $(\forall xs \in \text{Mapping.keys } a1\_map. \text{ case Mapping.lookup } a1\_map \text{ } xs \text{ of Some } tp' \Rightarrow tp' < tp) \wedge$   
 $(\forall tp' \in \text{Mapping.keys } a2\_map. \text{ case Mapping.lookup } a2\_map \text{ } tp' \text{ of Some } m \Rightarrow$   
 $\text{table } (\text{args\_n } \text{args}) (\text{args\_R } \text{args}) (\text{Mapping.keys } m) \wedge$   
 $(\forall xs \in \text{Mapping.keys } m. \text{ case Mapping.lookup } m \text{ } xs \text{ of Some } tstp \Rightarrow$   
 $tstp\_lt \text{ } tstp (\text{cur} - (\text{left } (\text{args\_ivl } \text{args}) - 1)) \text{ } tp \wedge (\text{isl } tstp \longleftrightarrow \text{left } (\text{args\_ivl } \text{args}) > 0))) \wedge$   
 $\text{length } \text{done} = \text{done\_length} \wedge \text{length } \text{done} + \text{len} = \text{length } \text{auxlist} \wedge$   
 $\text{rev } \text{done} = \text{map } \text{proj\_thd } (\text{take } (\text{length } \text{done}) \text{ } \text{auxlist}) \wedge$   
 $(\forall x \in \text{set } (\text{take } (\text{length } \text{done}) \text{ } \text{auxlist}). \text{check\_before } (\text{args\_ivl } \text{args}) \text{ } dt \text{ } x) \wedge$   
 $\text{sorted } (\text{map } \text{fst } \text{auxlist}) \wedge$   
 $\text{list\_all2 } (\lambda x y. \text{triple\_eq\_pair } x \text{ } y (\lambda tp'. \text{filter\_a1\_map } (\text{args\_pos } \text{args}) \text{ } tp' \text{ } a1\_map)$   
 $(\lambda ts' tp'. \text{filter\_a2\_map } (\text{args\_ivl } \text{args}) \text{ } ts' \text{ } tp' \text{ } a2\_map)) (\text{drop } (\text{length } \text{done}) \text{ } \text{auxlist})$   
 $(\text{zip } (\text{linearize } tss) [tp - \text{len}..<tp])$

**definition** *valid\_mmuaux* ::  $\text{args} \Rightarrow ts \Rightarrow 'a \text{ mmuaux} \Rightarrow 'a \text{ muaux} \Rightarrow$   
 $\text{bool}$  **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 (λ tstp 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 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map pos } tp' \text{ a1\_map})$ )
  ( $\lambda ts' tp'. \text{filter\_a2\_map } I ts' tp' \text{ a2\_map}$ )
  (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 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map pos } tp' \text{ a1\_map})$ )
  ( $\lambda ts' tp'. \text{filter\_a2\_map } I ts' tp' \text{ a2\_map}$ ) 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:  $\bigwedge ts' tp'. ts' \in \text{set } (\text{linearize } tss) \implies$ 
   $tp' \in \{tp - (len - 1)..<tp\} \implies \text{filter\_a2\_map } I ts' tp' \text{ a2\_map} =$ 
   $\text{filter\_a2\_map } I ts' tp' \text{ a2\_map}''$ 
proof (rule set_eqI, rule iffI)
  fix ts' tp' xs
  assume assms:  $ts' \in \text{set } (\text{linearize } tss)$ 
   $tp' \in \{tp - (len - 1)..<tp\}$   $xs \in \text{filter\_a2\_map } I ts' tp' \text{ a2\_map}$ 
obtain tp_bef m_bef tstp where defs:  $tp\_bef \leq tp'$   $\text{Mapping.lookup } a2\_map \text{ } tp\_bef = \text{Some } m\_bef$ 
   $\text{Mapping.lookup } m\_bef \text{ } xs = \text{Some } tstp$   $ts\_tp\_lt' ts' tp' \text{ } tstp$ 
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  $xs \in \text{filter\_a2\_map } I ts' tp' \text{ a2\_map}''$ 
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 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map } pos\ tp'\ a1\_map)$ )

```

```

    (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 ( $\lambda t. \text{enat } t + \text{right } I < \text{enat } nt$ ) tss)
have tss_list_takeWhile: tss_list = takeWhile ( $\lambda t. \text{enat } t + \text{right } I < \text{enat } nt$ ) (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 ( $\lambda t. \text{enat } t + \text{right } I < \text{enat } nt$ ) (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 ( $\lambda \text{aux } \_ . \text{eval\_step\_mmuaux } \text{aux}$ ) aux tss_list
by (cases foldl ( $\lambda \text{aux } \_ . \text{eval\_step\_mmuaux } \text{aux}$ ) 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 ( $\lambda \text{aux } \_ . \text{eval\_step\_mmuaux } \text{aux}$ ) aux tss_list) auxlist
  lin_ts_mmuaux (foldl ( $\lambda \text{aux } \_ . \text{eval\_step\_mmuaux } \text{aux}$ ) 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 ( $\lambda \text{aux } \_ . \text{eval\_step\_mmuaux } \text{aux}$ ) 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*  $\in$  *X* *then*  
 (*case Mapping.lookup m a of Some b*  $\Rightarrow$  *Some (f b)*  $\mid$  *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)  $\in$  X}*)  
 $\mid$  *None*  $\Rightarrow$  *if a*  $\in$  *fst 'X* *then Some (upd\_set' Mapping.empty d f {b. (a, b)  $\in$  X}*) *else None*)  
**by** (*simp add: Mapping.lookup.abs\_eq upd\_nested.abs\_eq*)

**lemma** *upd\_nested\_keys*:  $\text{Mapping.keys } (\text{upd\_nested } m \text{ d f } X) = \text{Mapping.keys } m \cup \text{fst } 'X$   
**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* =  $\bigcup ((\lambda \text{as. case Mapping.lookup } a1\_map \text{ (proj\_tuple } maskL \text{ as) of None} \Rightarrow$   
 (if  $\neg pos$  then  $\{(tp - len, as)\}$  else  $\{\}$ )  
 | *Some* *tp'*  $\Rightarrow$  if *pos* then  $\{(max (tp - len) tp', as)\}$   
 else  $\{(max (tp - len) (tp' + 1), as)\}$  ' *rel2*)  $\cup$  (if *left I* = 0 then  $\{tp\} \times rel2$  else  $\{\}$ );  
*a2\_map* = *Mapping.update* (*tp* + 1) *Mapping.empty*  
 (*upd\_nested* *a2\_map* *new\_tstp* (*max\_tstp* *new\_tstp*) *tmp*);  
*a1\_map* = (if *pos* then *Mapping.filter* ( $\lambda \_ . as \in rel1$ )  
 (*upd\_set* *a1\_map* ( $\lambda \_ . tp$ ) (*rel1* - *Mapping.keys* *a1\_map*)) else *upd\_set* *a1\_map* ( $\lambda \_ . tp$ ) *rel1*);  
*tss* = *append\_queue* *nt* *tss* in  
 (*tp* + 1, *tss*, *len* + 1, *maskL*, *maskR*, *a1\_map*, *a2\_map*, *done*, *done\_length*))

**lemma** *fst\_case*:  $(\lambda x. \text{fst } (\text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, y \text{ t } a1 \text{ a2}, z \text{ t } a1 \text{ a2}))) = \text{fst}$   
**by** *auto*

**lemma** *list\_all2\_in\_setE*:  $\text{list\_all2 } P \text{ xs } \text{ys} \Longrightarrow x \in \text{set } \text{xs} \Longrightarrow (\bigwedge y. y \in \text{set } \text{ys} \Longrightarrow P \text{ x } y \Longrightarrow Q) \Longrightarrow Q$   
**by** (*fastforce simp: list\_all2\_iff set\_zip in\_set\_conv\_nth*)

**lemma** *list\_all2\_zip*:  $\text{list\_all2 } (\lambda x \ y. \text{triple\_eq\_pair } x \ y \ f \ g) \text{ xs } (\text{zip } \text{ys } \text{zs}) \Longrightarrow$   
 $(\bigwedge y. y \in \text{set } \text{ys} \Longrightarrow Q \ y) \Longrightarrow x \in \text{set } \text{xs} \Longrightarrow Q \ (\text{fst } x)$   
**by** (*auto simp: in\_set\_zip elim!: list\_all2\_in\_setE triple\_eq\_pair.elims*)

**lemma** *list\_appendE*:  $\text{xs} = \text{ys} @ \text{zs} \Longrightarrow x \in \text{set } \text{xs} \Longrightarrow$   
 $(x \in \text{set } \text{ys} \Longrightarrow P) \Longrightarrow (x \in \text{set } \text{zs} \Longrightarrow P) \Longrightarrow P$   
**by** *auto*

**lemma** *take\_takeWhile*:  $n \leq \text{length } \text{ys} \Longrightarrow$   
 $(\bigwedge y. y \in \text{set } (\text{take } n \ \text{ys}) \Longrightarrow P \ y) \Longrightarrow$   
 $(\bigwedge y. y \in \text{set } (\text{drop } n \ \text{ys}) \Longrightarrow \neg P \ y) \Longrightarrow$   
 $\text{take } n \ \text{ys} = \text{takeWhile } P \ \text{ys}$

**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*:  $nt \geq cur$   
**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 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map pos } tp' \text{ a1\_map})$ )
( $\lambda ts' tp'. \text{filter\_a2\_map } I \text{ ts' } tp' \text{ a2\_map}$ ) (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:  $\bigwedge t. t \in \text{set } (\text{map fst } (\text{take } (\text{length done}) \text{ auxlist})) \implies t \leq \text{nt}$ 
using set_take_auxlist
by auto (meson add_increasing2 enat_ord_simps(1) le_cases not_less zero_le)
have map_fst_auxlist_drop:  $\bigwedge t. t \in \text{set } (\text{map fst } (\text{drop } (\text{length done}) \text{ auxlist})) \implies t \leq \text{nt}$ 
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)
  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  $tstp\_def[\text{unfolded } m'\_alt \ upd\_set'\_lookup] \ \text{new\_tstp\_lt\_isl}$ 
        by (auto split: if_splits)
    next
      case (Some  $tstp'$ )
      show ?thesis
      proof (cases  $xs \in \{b. (tp', b) \in tmp\}$ )
        case True
        then have  $tstp\_eq: tstp = \max\_tstp \ \text{new\_tstp } \ tstp'$ 
          using  $tstp\_def[\text{unfolded } m'\_alt \ upd\_set'\_lookup] \ \text{Some}$ 
          by auto
        show ?thesis
          using  $\text{lookup\_a2\_map}'[OF \ m\_def \ \text{Some}] \ \text{new\_tstp\_lt\_isl}$ 
          by (auto simp add:  $tstp\_lt\_def \ tstp\_eq$  split: sum.splits)
      next
        case False
        then show ?thesis
          using  $tstp\_def[\text{unfolded } m'\_alt \ upd\_set'\_lookup] \ \text{lookup\_a2\_map}'[OF \ m\_def \ \text{Some}] \ \text{Some}$ 
          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  $map\_eq: \text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } pos \text{ then } \text{join } a1 \ \text{True } rel1 \ \text{else } a1 \cup rel1,$ 
     $\text{if } mem \ (nt - t) \ I \ \text{then } a2 \cup \text{join } rel2 \ pos \ a1 \ \text{else } a2)) \ (\text{drop } (\text{length } done) \ \text{auxlist}) =$ 
     $\text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } pos \text{ then } \text{join } a1 \ \text{True } rel1 \ \text{else } a1 \cup rel1,$ 
     $\text{if } \text{left } I \leq nt - t \ \text{then } a2 \cup \text{join } rel2 \ pos \ a1 \ \text{else } a2)) \ (\text{drop } (\text{length } done) \ \text{auxlist})$ 
    using  $set\_drop\_auxlist\_cong$  by auto
  have  $\text{drop } (\text{length } done) \ (\text{update\_until } args \ rel1 \ rel2 \ nt \ \text{auxlist}) =$ 
     $\text{map } (\lambda x. \text{case } x \text{ of } (t, a1, a2) \Rightarrow (t, \text{if } pos \text{ then } \text{join } a1 \ \text{True } rel1 \ \text{else } a1 \cup rel1,$ 
     $\text{if } mem \ (nt - t) \ I \ \text{then } a2 \cup \text{join } rel2 \ pos \ a1 \ \text{else } a2)) \ (\text{drop } (\text{length } done) \ \text{auxlist}) \ @$ 
     $[(nt, rel1, \text{if } \text{left } I = 0 \ \text{then } rel2 \ \text{else } \text{empty\_table})]$ 
    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  $\text{list\_all2\_map1}$ 
using  $\text{list\_all2\_triple}$ 
proof ( $\text{rule list.rel\_mono\_strong}$ )
fix  $\text{tri pair}$ 
assume  $\text{tri\_pair\_in: tri } \in \text{set } (\text{drop } (\text{length done}) \text{ auxlist})$ 
 $\text{pair } \in \text{set } (\text{zip } (\text{linearize tss}) [tp - \text{len}..<tp])$ 
obtain  $t \text{ } a1 \text{ } a2$  where  $\text{tri\_def: tri} = (t, a1, a2)$ 
by ( $\text{cases tri}$ ) auto
obtain  $ts' \text{ } tp'$  where  $\text{pair\_def: pair} = (ts', tp')$ 
by ( $\text{cases pair}$ ) auto
assume  $\text{triple\_eq\_pair tri pair } (\lambda tp'. \text{filter\_a1\_map pos } tp' \text{ } a1\_map)$ 
 $(\lambda ts' tp'. \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map)$ 
then have  $\text{eqs: } t = ts' \text{ } a1 = \text{filter\_a1\_map pos } tp' \text{ } a1\_map$ 
 $a2 = \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map$ 
unfolding  $\text{tri\_def pair\_def}$  by auto
have  $tp'_{\text{ge}}: tp' \geq tp - \text{len}$ 
using  $\text{tri\_pair\_in}(2)$  unfolding  $\text{pair\_def}$ 
by ( $\text{auto elim: in\_set\_zipE}$ )
have  $tp'_{\text{lt\_tp}}: tp' < tp$ 
using  $\text{tri\_pair\_in}(2)$  unfolding  $\text{pair\_def}$ 
by ( $\text{auto elim: in\_set\_zipE}$ )
have  $ts'_{\text{in\_lin\_tss}}: ts' \in \text{set } (\text{linearize tss})$ 
using  $\text{tri\_pair\_in}(2)$  unfolding  $\text{pair\_def}$ 
by ( $\text{auto elim: in\_set\_zipE}$ )
then have  $ts'_{\text{nt}}: ts' \leq nt$ 
using  $\text{valid\_shift\_aux}(1)$   $\text{assms}(4)$  by auto
then have  $t_{\text{nt}}: t \leq nt$ 
unfolding  $\text{eqs}(1)$  .
have  $\text{table } n \text{ } L \text{ } (\text{Mapping.keys } a1\_map)$ 
using  $\text{valid\_shift\_aux}$  unfolding  $n\_def \text{ } L\_def$  by auto
then have  $a1\_tab: \text{table } n \text{ } L \text{ } a1$ 
unfolding  $\text{eqs}(2)$   $\text{filter\_a1\_map\_def}$  by ( $\text{auto simp add: table\_def}$ )
note  $\text{tabR} = \text{tabs}(2)[\text{unfolded } n\_def[\text{symmetric}] \text{ } R\_def[\text{symmetric}]$ 
have  $\text{join\_rel2\_assms: } L \subseteq R \text{ } \text{maskL} = \text{join\_mask } n \text{ } L$ 
using  $\text{valid\_shift\_aux}$  unfolding  $n\_def \text{ } L\_def \text{ } R\_def$  by auto
have  $\text{join\_rel2\_eq: join rel2 pos } a1 = \{xs \in \text{rel2. } \text{proj\_tuple\_in\_join pos } \text{maskL } xs \text{ } a1\}$ 
using  $\text{join\_sub}[OF \text{ join\_rel2\_assms}(1) \text{ } a1\_tab \text{ } \text{tabR}] \text{ join\_rel2\_assms}(2)$  by auto
have  $\text{filter\_sub\_a2: } \bigwedge xs \text{ } m' \text{ } tp'' \text{ } tstp. tp'' \leq tp' \implies$ 
 $\text{Mapping.lookup } a2\_map' \text{ } tp'' = \text{Some } m' \implies \text{Mapping.lookup } m' \text{ } xs = \text{Some } tstp \implies$ 
 $ts\_tp\_lt' \text{ } ts' \text{ } tp' \text{ } tstp \implies (tstp = \text{new\_tstp} \implies \text{False}) \implies$ 
 $xs \in \text{filter\_a2\_map } I \text{ } ts' \text{ } tp' \text{ } a2\_map' \implies xs \in a2$ 
proof –
fix  $xs \text{ } m' \text{ } tp'' \text{ } tstp$ 
assume  $m'_{\text{def}}: tp'' \leq tp' \text{ } \text{Mapping.lookup } a2\_map' \text{ } tp'' = \text{Some } m'$ 
 $\text{Mapping.lookup } m' \text{ } xs = \text{Some } tstp \text{ } ts\_tp\_lt' \text{ } ts' \text{ } tp' \text{ } tstp$ 
have  $tp''_{\text{neq}}: tp + 1 \neq tp''$ 
using  $\text{le\_less\_trans}[OF \text{ } m'_{\text{def}}(1) \text{ } tp'_{\text{lt\_tp}}]$  by auto
assume  $\text{new\_tstp\_False: } tstp = \text{new\_tstp} \implies \text{False}$ 
show  $xs \in a2$ 
proof ( $\text{cases Mapping.lookup } a2\_map' \text{ } tp''$ )
case None
then have  $m'_{\text{alt}}: m' = \text{upd\_set}' \text{ } \text{Mapping.empty } \text{new\_tstp} \text{ } (\text{max\_tstp } \text{new\_tstp})$ 
 $\{b. (tp'', b) \in \text{tmp}\}$ 
using  $m'_{\text{def}}(2)[\text{unfolded } a2\_map'_{\text{def}} \text{ } \text{Mapping.lookup\_update\_neq}[OF \text{ } tp''_{\text{neq}}]$ 

```

```

    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  $xs\_props: xs \in \text{rel2 } xs \notin a2 \text{ } proj\_tuple\_in\_join \text{ } pos \text{ } maskL \text{ } xs \text{ } a1$ 
    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 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  $tp'''\_def: tp''' \leq tp'$ 
     $\text{Mapping.lookup } a1\_map \text{ } (proj\_tuple \text{ } maskL \text{ } xs) = \text{Some } tp'''$ 
    using  $eqs(2)[\text{unfolded } filter\_a1\_map\_def] \text{ } xs\_props(3)[\text{unfolded } proj\_tuple\_in\_join\_def]$ 
    by (auto dest: 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  $wtp\_in \text{ } a2\_map\_keys \text{ } Mapping\_keys\_dest$  by fastforce
  obtain  $m'$  where  $m'\_def: \text{Mapping.lookup } a2\_map' \text{ } wtp = \text{Some } m'$ 
    using  $wtp\_in \text{ } a2\_map'\_keys \text{ } Mapping\_keys\_dest$  by fastforce
  have  $m'\_alt: m' = \text{upd\_set}' \text{ } m \text{ } new\_tstp \text{ } (max\_tstp \text{ } new\_tstp) \{b. (wtp, b) \in tmp\}$ 
    using  $m'\_def[\text{unfolded } a2\_map'\_def \text{ } Mapping.lookup\_update\_neq[OF \text{ } wtp\_neq]]$ 
     $\text{upd\_nested\_lookup } m\_def]$  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  $m'\_alt \text{ } upd\_set'\_lookup$  None by auto
  then show ?thesis
    unfolding  $filter\_a2\_map\_def$  using  $wtp\_le \text{ } m'\_def \text{ } ts\_tp\_lt'\_new\_tstp$  by auto
next
  case (Some tstp')
  have  $\text{Mapping.lookup } m' \text{ } xs = \text{Some } (max\_tstp \text{ } new\_tstp \text{ } tstp')$ 
    using  $wtp\_xs\_in$  unfolding  $m'\_alt \text{ } upd\_set'\_lookup$  Some by auto
  moreover have  $ts\_tp\_lt' \text{ } ts' \text{ } tp' \text{ } (max\_tstp \text{ } new\_tstp \text{ } tstp')$ 
using  $max\_tstp\_intro'''' \text{ } ts\_tp\_lt'\_new\_tstp \text{ } lookup\_a2\_map'[\text{OF } m\_def \text{ } \text{Some}] \text{ } new\_tstp\_lt\_isl$ 
by auto
  ultimately show ?thesis
    using  $lookup\_a2\_map'[\text{OF } m\_def \text{ } \text{Some}] \text{ } wtp\_le \text{ } m'\_def$ 
    unfolding  $filter\_a2\_map\_def$  by auto
  qed
next
  case False
  show ?thesis
proof (cases  $\text{Mapping.lookup } a1\_map \text{ } (proj\_tuple \text{ } maskL \text{ } xs)$ )
  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 ( $\lambda x y. \text{triple\_eq\_pair } x y (\lambda tp'. \text{filter\_a1\_map pos } tp' \text{ a1\_map}')$ )
    ( $\lambda ts' tp'. \text{filter\_a2\_map } I ts' tp' \text{ a2\_map}'$ )
    (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 ( $\lambda x y. \text{triple\_eq\_pair } x y f g$ ) xs (zip ys zs)  $\implies$ 
  ( $\bigwedge y. y \in \text{set } ys \implies \neg \text{enat } y + \text{right } I < nt$ )  $\implies x \in \text{set } xs \implies \neg \text{check\_before } I nt x$ 
  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
     $\bigwedge ts. ts \in \text{set } (\text{linearize } tss) \implies \neg \text{enat } ts + \text{right } (\text{args\_ivl } args) < \text{enat } nt$ 
    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:  $\bigwedge x. x \in \text{set } (\text{take } (\text{length } done) \text{ auxlist}) \implies \text{check\_before } I nt x$ 
    using valid_shift_aux unfolding I_def by auto
  have set_drop_auxlist:  $\bigwedge x. x \in \text{set } (\text{drop } (\text{length } done) \text{ auxlist}) \implies \neg \text{check\_before } I nt x$ 
    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:  $f \text{ ' } Option.these X = Option.these (map\_option f \text{ ' } X)$

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*:  $\text{meval } n \ t \ \text{db} \ (\text{MPred } e \ ts) =$   
 (case Mapping.lookup db e of None  $\Rightarrow$   $\{\}$  | Some Xs  $\Rightarrow$  map  $(\lambda X. \bigcup v \in X.$   
 (set\_option (map\_option  $(\lambda f. \text{Table.tabulate } f \ 0 \ n) \ (\text{match } ts \ v))))$  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  $(\bigcup n \in \text{set} \ (\text{map } \text{fst } t). \ (\lambda v. \ (n, \ v)))$  ‘the (map\_of t n)’

**definition** *rbt\_fold* ::  $\_ \Rightarrow$  event\_data tuple set\_rbt  $\Rightarrow$   $\_ \Rightarrow$   $\_$  **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*:  $\text{mono } g \ \bigwedge X. X \subseteq C \implies g \ X \subseteq C$  *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  $\lambda S. S = \{\} \vee r \in S \ \lambda S. g \ S \neq S \ g \ \lambda S. h \ S \neq S$  *insert*  
*r h \{\}*, *symmetric*])  
 (*auto 4 4 dest: while\_option\_stop*[of  $\lambda S. g \ S \neq S \ g \ \{\}$ ])  
**qed**

**definition** *RPDs\_aux* = *saturate*  $(\lambda S. S \cup \bigcup (RPD \ ' S))$

**lemma** *RPDs\_aux\_code[code]*:  
*RPDs\_aux* S = (let S' = S  $\cup$  Set.bind S RPD in if S'  $\subseteq$  S then S else *RPDs\_aux* S')  
**unfolding** *RPDs\_aux\_def* *bind\_UNION*  
**by** (subst *saturate\_code*) *auto*

**lemma** *RPDs\_code[code]*: *RPDs* r = *RPDs\_aux*  $\{r\}$   
**unfolding** *RPDs\_aux\_def* *RPDs\_code*  
**by** (rule *saturate\_commute*[**where** C=*RPDs* r])  
 (*auto 0 3 simp: mono\_def subset\_singleton\_iff RPDs\_refl RPDs\_trans finite\_RPDs*)

**definition** *LPDs\_aux* = *saturate*  $(\lambda S. S \cup \bigcup (LPD \ ' S))$

**lemma** *LPDs\_aux\_code[code]*:  
*LPDs\_aux* S = (let S' = S  $\cup$  Set.bind S LPD in if S'  $\subseteq$  S then S else *LPDs\_aux* S')

**unfolding** *LPDs\_aux\_def* *bind\_UNION*  
**by** (*subst saturate\_code*) *auto*

**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*  $\longleftrightarrow$  *Set.is\_empty X*  
*empty\_table = X*  $\longleftrightarrow$  *Set.is\_empty X*  
*set\_eq X empty\_table*  $\longleftrightarrow$  *Set.is\_empty X*  
*set\_eq empty\_table X*  $\longleftrightarrow$  *Set.is\_empty X*  
*X = (set\_empty impl)*  $\longleftrightarrow$  *Set.is\_empty X*  
(*set\_empty impl*) = *X*  $\longleftrightarrow$  *Set.is\_empty X*  
*set\_eq X (set\_empty impl)*  $\longleftrightarrow$  *Set.is\_empty X*  
*set\_eq (set\_empty impl) X*  $\longleftrightarrow$  *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*  $\Rightarrow$  *Code.abort (STR "tabulate RBT\_Mapping: ccompare = None")* ( $\lambda\_.$  *Monitor.mrtabulate (xs :: mregeX list) f*)  
 $\mid$   $\Rightarrow$  *RBT\_Mapping (RBT\_Mapping2.bulkload (List.map\_filter ( $\lambda k.$  let *fk = f k* in if *fk = empty\_table* then *None* else *Some (k, fk)*) xs))*)  
**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*  $\Rightarrow$  *Code.abort (STR "combine RBT\_Mapping: ccompare = None")* ( $\lambda\_.$  *Mapping.combine f (RBT\_Mapping t) (RBT\_Mapping u)*)  
 $\mid$  *Some \_*  $\Rightarrow$  *RBT\_Mapping (RBT\_Mapping2.join ( $\lambda\_.$  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\_eq1*) (*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 ( $\lambda a.$  Mapping.update a (f a)) m A*  
**proof** –  
**interpret** *comp\_fun\_idem*  $\lambda a.$  *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*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a,* (*'a,* *'b*) *mapping*) *comp\_fun\_idem*  
**is**  $\lambda f a m.$  *Mapping.update a (f a) m*  
**by** *unfold\_locales (transfer; auto simp: fun\_eq\_iff)+*

**lemma** *upd\_set\_code*[*code*]:  
*upd\_set m f A = (if finite A then set\_fold\_cfi (upd\_cfi f) m A else Code.abort (STR "upd\_set: infinite")*  
( $\lambda\_.$  *upd\_set m f A*)

by (transfer fixing: m) (auto simp: upd\_set\_fold)

**lemma** *lexordp\_eq\_code*[code]:  $\text{lexordp\_eq } xs \ ys \iff (\text{case } xs \text{ of } [] \Rightarrow \text{True} \mid x \# xs \Rightarrow (\text{case } ys \text{ of } [] \Rightarrow \text{False} \mid y \# ys \Rightarrow \text{if } x < y \text{ then True else if } x > y \text{ then False else lexordp\_eq } xs \ ys))$   
 by (subst *lexordp\_eq\_simps*) (auto split: list.split)

**definition** *filter\_set* m X t = *Mapping.filter* (*filter\_cond* X m t) m

**declare** *shift\_end\_simps*[folded *filter\_set\_def*, code]

**lemma** *upd\_set'\_empty*[simp]:  $\text{upd\_set}' \ m \ d \ f \ \{\} = m$   
 by (rule *mapping\_eqI*) (auto simp add: *upd\_set'\_lookup*)

**lemma** *upd\_set'\_insert*:  $d = f \ d \implies (\bigwedge x. f \ (f \ x) = f \ x) \implies \text{upd\_set}' \ m \ d \ f \ (\text{insert } x \ A) =$   
 (let  $m' = (\text{upd\_set}' \ m \ d \ f \ A)$  in case *Mapping.lookup*  $m' \ x$  of *None*  $\Rightarrow$  *Mapping.update*  $x \ d \ m'$   
 | *Some v*  $\Rightarrow$  *Mapping.update*  $x \ (f \ v) \ m'$ )  
 by (rule *mapping\_eqI*) (auto simp: *upd\_set'\_lookup Mapping.lookup\_update'* split: option.splits)

**lemma** *upd\_set'\_aux1*:  $\text{upd\_set}' \ \text{Mapping.empty} \ d \ f \ \{b. b = k \vee (a, b) \in A\} =$   
*Mapping.update*  $k \ d \ (\text{upd\_set}' \ \text{Mapping.empty} \ d \ f \ \{(a, b) \in A\})$   
 by (rule *mapping\_eqI*) (auto simp add: *Let\_def upd\_set'\_lookup Mapping.lookup\_update'*  
*Mapping.lookup\_empty split: option.splits*)

**lemma** *upd\_set'\_aux2*:  $\text{Mapping.lookup} \ m \ k = \text{None} \implies \text{upd\_set}' \ m \ d \ f \ \{b. b = k \vee (a, b) \in A\} =$   
*Mapping.update*  $k \ d \ (\text{upd\_set}' \ m \ d \ f \ \{(a, b) \in A\})$   
 by (rule *mapping\_eqI*) (auto simp add: *upd\_set'\_lookup Mapping.lookup\_update'* split: option.splits)

**lemma** *upd\_set'\_aux3*:  $\text{Mapping.lookup} \ m \ k = \text{Some } v \implies \text{upd\_set}' \ m \ d \ f \ \{b. b = k \vee (a, b) \in A\} =$   
*Mapping.update*  $k \ (f \ v) \ (\text{upd\_set}' \ m \ d \ f \ \{(a, b) \in A\})$   
 by (rule *mapping\_eqI*) (auto simp add: *upd\_set'\_lookup Mapping.lookup\_update'* split: option.splits)

**lemma** *upd\_set'\_aux4*:  $k \notin \text{fst } 'A \implies \text{upd\_set}' \ \text{Mapping.empty} \ d \ f \ \{(k, b) \in A\} = \text{Mapping.empty}$   
 by (rule *mapping\_eqI*) (auto simp add: *upd\_set'\_lookup Mapping.lookup\_update'* *Domain.DomainI fst\_eq\_Domain*  
*split: option.splits*)

**lemma** *upd\_nested\_empty*[simp]:  $\text{upd\_nested} \ m \ d \ f \ \{\} = m$   
 by (rule *mapping\_eqI*) (auto simp add: *upd\_nested\_lookup split: option.splits*)

**definition** *upd\_nested\_step* ::  $'c \Rightarrow ('c \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow ('a, ('b, 'c) \text{ mapping}) \text{ mapping} \Rightarrow$   
 $('a, ('b, 'c) \text{ mapping}) \text{ mapping}$  where  
 $\text{upd\_nested\_step} \ d \ f \ x \ m = (\text{case } x \text{ of } (k, k') \Rightarrow$   
 (case *Mapping.lookup*  $m \ k$  of *Some m'*  $\Rightarrow$   
 (case *Mapping.lookup*  $m' \ k'$  of *Some v*  $\Rightarrow$  *Mapping.update*  $k \ ( \text{Mapping.update } k' \ (f \ v) \ m' ) \ m$   
 | *None*  $\Rightarrow$  *Mapping.update*  $k \ ( \text{Mapping.update } k' \ d \ m' ) \ m$   
 | *None*  $\Rightarrow$  *Mapping.update*  $k \ ( \text{Mapping.update } k' \ d \ \text{Mapping.empty} ) \ m$ ))

**lemma** *upd\_nested\_insert*:  
 $d = f \ d \implies (\bigwedge x. f \ (f \ x) = f \ x) \implies \text{upd\_nested} \ m \ d \ f \ (\text{insert } x \ A) =$   
*upd\_nested\_step*  $d \ f \ x \ (\text{upd\_nested} \ m \ d \ f \ A)$   
**unfolding** *upd\_nested\_step\_def*  
**using** *upd\_set'\_aux1*[of  $d \ f \ \_ \ A$ ] *upd\_set'\_aux2*[of  $\_ \ \_ \ d \ f \ \_ \ A$ ] *upd\_set'\_aux3*[of  $\_ \ \_ \ \_ \ d \ f \ \_ \ A$ ]  
*upd\_set'\_aux4*[of  $\_ \ A \ d \ f$ ]  
 by (auto simp add: *Let\_def upd\_nested\_lookup upd\_set'\_lookup Mapping.lookup\_update'*  
*Mapping.lookup\_empty split: option.splits prod.splits if\_splits intro!: mapping\_eqI*)

**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* ::

$ts + tp \Rightarrow ('a \times 'b, ('a, ('b, ts + tp)\ mapping)\ mapping)\ comp\_fun\_idem$

**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]*:  $filter\_set\ m\ \{\} = m$

**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 | \_ \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") (λ\_. 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 ⇒ Code.abort (STR "filter RBT\_Mapping: ccompare = None")  
 (λ\_. Mapping.filter P (RBT\_Mapping t))  
 | Some \_ ⇒ RBT\_Mapping (RBT\_Mapping2.filter (case\_prod P) t))  
 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 ¬pos ∧ finite A ∧ card A < Mapping.size m then set\_fold\_cfi filter\_not\_in\_cfi m A  
 else Mapping.filter (join\_filter\_cond pos A) m)  
**unfolding** *filter\_join\_def*  
 by (transfer fixing: m) (use filter\_join\_False in ⟨auto simp add: filter\_join\_def⟩)

**definition** *set\_minus* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  'a set **where**  
*set\_minus* X Y = X - Y

**lift\_definition** *remove\_cfi* :: ('a, 'a set) *comp\_fun\_idem*  
**is**  $\lambda b a. a - \{b\}$   
**by** *unfold\_locales auto*

**lemma** *set\_minus\_finite*:  
**assumes** *fin*: *finite* Y  
**shows** *set\_minus* X Y = *Finite\_Set.fold* ( $\lambda a X. X - \{a\}$ ) X Y

**proof** -  
**interpret** *comp\_fun\_idem*  $\lambda a X. X - \{a\}$   
**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  $\wedge$  *card* Y < *card* X *then* *set\_fold\_cfi* *remove\_cfi* X Y *else* X - Y)  
**by** *transfer* (*use* *set\_minus\_finite* **in**  $\langle$ *auto simp add*: *set\_minus\_def $\rangle$ )*

**declare** *bin\_join.simps*[*folded* *set\_minus\_def*, *code*]

**definition** *remove\_Union* **where**  
*remove\_Union* A X B = A - ( $\bigcup x \in X. B x$ )

**lemma** *remove\_Union\_finite*:  
**assumes** *finite* X  
**shows** *remove\_Union* A X B = *Finite\_Set.fold* ( $\lambda x A. A - B x$ ) A X

**proof** -  
**interpret** *comp\_fun\_idem*  $\lambda x A. A - B x$   
**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  $\Rightarrow$  'b set)  $\Rightarrow$  ('a, 'b set) *comp\_fun\_idem* **is**  $\lambda B x A. A - B x$   
**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 - ( $\bigcup x \in X. B x$ ))  
**by** (*transfer* *fixing*: A X B) (*use* *remove\_Union\_finite*[*of* X A B] **in**  $\langle$ *auto simp add*: *remove\_Union\_def $\rangle$ )*

**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  $\times$  event\_data list set) list  $\Rightarrow$  (String.literal, event\_data list set list) alist **is**

$\lambda t. List.map\_filter (\lambda(p, X). \text{if } X = \{\} \text{ then None else Some } (p, [X])) (AList.clearjunk t)$   
**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* ( $\lambda(a, b). \text{if } P b \text{ then None else Some } (f a b)$ ) xs =  
*map* ( $\lambda(a, b). f a b$ ) (*filter* ( $\lambda x. \neg P (\text{snd } x)$ ) 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 (λ(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.