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

Thibault Dardinier Lukas Heimes Martin Raszyk Joshua Schneider
Dmitriy Traytel

February 23, 2021

Abstract

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

Contents

1 Code adaptation for IEEE double-precision floats **2**
1.1 copysign . . . 2
1.2 Additional lemmas about generic floats . . . 2
1.3 Doubles with a unified NaN value . . . 4
1.4 Linear ordering . . . 6
1.4.1 Code setup . . . 8

2 Event parameters **9**

3 Regular expressions **11**

4 Metric first-order dynamic logic **14**
4.1 Formulas and satisfiability . . . 15
4.1.1 Syntax . . . 15
4.1.2 Future reach . . . 17
4.1.3 Semantics . . . 18
4.2 Past-only formulas . . . 20
4.3 Safe formulas . . . 20
4.4 Slicing traces . . . 22
4.5 Translation to n-ary conjunction . . . 23

5 Optimized relational join **25**
5.1 Binary join . . . 25
5.2 Multi-way join . . . 26

6 Generic monitoring algorithm **30**
6.1 Monitorable formulas . . . 30
6.2 Handling regular expressions . . . 31
6.2.1 LPD . . . 33
6.2.2 RPD . . . 35
6.3 The executable monitor . . . 36
6.4 Verdict delay . . . 45
6.5 Specification . . . 48
6.6 Correctness . . . 49
6.6.1 Invariants . . . 49
6.6.2 Initialisation . . . 54
6.6.3 Evaluation . . . 55
6.6.4 Monitor step . . . 70
6.6.5 Monitor function . . . 71
6.7 Collected correctness results . . . 72

7 Efficient implementation of temporal operators **73**
7.1 Optimized queue data structure . . . 73
7.2 Optimized data structure for Since . . . 76
7.3 Optimized data structure for Until . . . 83

8 Instantiation of the generic algorithm and code setup **88**

1 Code adaptation for IEEE double-precision floats

1.1 copysign

lift_definition *copysign* :: ('*e*, '*f*) *float* ⇒ ('*e*, '*f*) *float* ⇒ ('*e*, '*f*) *float* **is**
λ(_, *e*::'*e word*, *f*::'*f word*) (*s*::*1 word*, _, _). (*s*, *e*, *f*) ⟨*proof*⟩

lemma *is_nan_copysign*[*simp*]: *is_nan* (*copysign x y*) ⟷ *is_nan x*
⟨*proof*⟩

1.2 Additional lemmas about generic floats

lemma *is_nan_some_nan*[*simp*]: *is_nan* (*some_nan* :: ('*e*, '*f*) *float*)
⟨*proof*⟩

lemma *not_is_nan_0*[*simp*]: ¬ *is_nan 0*
⟨*proof*⟩

lemma *not_is_nan_1*[*simp*]: ¬ *is_nan 1*
⟨*proof*⟩

lemma *is_nan_plus*: *is_nan x* ∨ *is_nan y* ⟹ *is_nan* (*x* + *y*)
⟨*proof*⟩

lemma *is_nan_minus*: *is_nan x* ∨ *is_nan y* ⟹ *is_nan* (*x* − *y*)
⟨*proof*⟩

lemma *is_nan_times*: *is_nan x* ∨ *is_nan y* ⟹ *is_nan* (*x* ∗ *y*)
⟨*proof*⟩

lemma *is_nan_divide*: *is_nan x* ∨ *is_nan y* ⟹ *is_nan* (*x* / *y*)
⟨*proof*⟩

lemma *is_nan_float_sqrt*: *is_nan x* $\Longrightarrow$ *is_nan* (*float_sqrt x*)
⟨*proof*⟩

lemma *nan_fcompare*: *is_nan x* $\vee$ *is_nan y* $\Longrightarrow$ *fcompare x y* = *Und*
⟨*proof*⟩

lemma *nan_not_le*: *is_nan x* $\vee$ *is_nan y* $\Longrightarrow$ $\neg$ *x* $\leq$ *y*
⟨*proof*⟩

lemma *nan_not_less*: *is_nan x* $\vee$ *is_nan y* $\Longrightarrow$ $\neg$ *x* $<$ *y*
⟨*proof*⟩

lemma *nan_not_zero*: *is_nan x* $\Longrightarrow$ $\neg$ *is_zero x*
⟨*proof*⟩

lemma *nan_not_infinity*: *is_nan x* $\Longrightarrow$ $\neg$ *is_infinity x*
⟨*proof*⟩

lemma *zero_not_infinity*: *is_zero x* $\Longrightarrow$ $\neg$ *is_infinity x*
⟨*proof*⟩

lemma *zero_not_nan*: *is_zero x* $\Longrightarrow$ $\neg$ *is_nan x*
⟨*proof*⟩

lemma *minus_one_power_one_word*: (−1 :: *real*) ^ *unat* (*x* :: 1 *word*) = (*if unat x* = 0 *then* 1 *else* −1)
⟨*proof*⟩

definition *valofn* :: ('*e*, '*f*) *float* ⇒ *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* ⇒ *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*)
⟨*proof*⟩

lemma *fraction_less_2p*: *fraction* (*x* :: ('*e*, '*f*) *float*) < 2^*LENGTH*('*f*)
⟨*proof*⟩

lemma *valofn_ge_0*: 0 ≤ *valofn x*
⟨*proof*⟩

lemma *valofn_ge_2p*: 2^*exponent* (*x* :: ('*e*, '*f*) *float*) / 2^*bias TYPE*(('*e*, '*f*) *float*) ≤ *valofn x*
⟨*proof*⟩

lemma *valofn_less_2p*:
fixes *x* :: ('*e*, '*f*) *float*
assumes *exponent x* < *e*
shows *valofn x* < 2^*e* / 2^*bias TYPE*(('*e*, '*f*) *float*)
⟨*proof*⟩

lemma *valofd_ge_0*: 0 ≤ *valofd x*
⟨*proof*⟩

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

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

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

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

lemma *is_zero_valof_conv*: *is_zero x* ⟷ *valof x* = *0*
⟨*proof*⟩

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

lemma *sign_gt_0_conv*: *0* < *sign x* ⟷ *sign x* = *1*
⟨*proof*⟩

lemma *valof_eq*:
assumes ¬ *is_zero x* ∨ ¬ *is_zero y*
shows *valof x* = *valof y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *zero_fcompare*: *is_zero x* ⟹ *is_zero y* ⟹ *fcompare x y* = *ccode.Eq*
⟨*proof*⟩

1.3 Doubles with a unified NaN value

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

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

lift_definition *zero_double* :: *double* **is** *0* ⟨*proof*⟩
lift_definition *one_double* :: *double* **is** *1* ⟨*proof*⟩

lift_definition *plus_double* :: *double* ⇒ *double* ⇒ *double* **is** *plus*
⟨*proof*⟩

lift_definition *minus_double* :: *double* ⇒ *double* ⇒ *double* **is** *minus*
⟨*proof*⟩

lift_definition *uminus_double* :: *double* ⇒ *double* **is** *uminus*
⟨*proof*⟩

lift_definition *times_double* :: *double* ⇒ *double* ⇒ *double* **is** *times*
⟨*proof*⟩

lift_definition *less_eq_double* :: *double* ⇒ *double* ⇒ *bool* **is** (≤)
⟨*proof*⟩

lift_definition *less_double* :: *double* ⇒ *double* ⇒ *bool* **is** (<)
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *double* :: *inverse*
begin

lift_definition *divide_double* :: *double* ⇒ *double* ⇒ *double* **is** *divide*
⟨*proof*⟩

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

instance ⟨*proof*⟩

end

lift_definition *sqrt_double* :: *double* ⇒ *double* **is** *float_sqrt*
⟨*proof*⟩

no_notation *plus_infinity* (∞)

lift_definition *infinity* :: *double* **is** *plus_infinity* ⟨*proof*⟩

lift_definition *nan* :: *double* **is** *some_nan* ⟨*proof*⟩

lift_definition *is_zero* :: *double* ⇒ *bool* **is** *IEEE.is_zero*
⟨*proof*⟩

lift_definition *is_infinite* :: *double* ⇒ *bool* **is** *IEEE.is_infinity*
⟨*proof*⟩

lift_definition *is_nan* :: *double* ⇒ *bool* **is** *IEEE.is_nan*
⟨*proof*⟩

lemma *is_nan_conv*: *is_nan x* ⟷ *x* = *nan*
⟨*proof*⟩

lift_definition *copysign_double* :: *double* ⇒ *double* ⇒ *double* **is**
λ*x y*. *if IEEE.is_nan y then some_nan else copysign x y*
⟨*proof*⟩

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

lift_definition *fcompare_double* :: *double* ⇒ *double* ⇒ *ccode* **is** *fcompare*
⟨*proof*⟩

lemma *nan_fcompare_double*: *is_nan x* ∨ *is_nan y* ⟹ *fcompare_double x y* = *Und*
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *compare_double_simps* = *compare_double_less compare_double_eq compare_double_greater*

lemma *compare_double_le_0*: *compare_double x y* ≤ *0* ⟷
is_nan x ∨ *fcompare_double x y* ∈ {*ccode.Eq*, *ccode.Lt*}
⟨*proof*⟩

lift_definition *double_of_integer* :: *integer* ⇒ *double* **is**
λ*x*. *zerosign 0* (*intround To_nearest* (*int_of_integer x*)) ⟨*proof*⟩

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

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

lift_definition *integer_of_double* :: *double* ⇒ *integer* **is**
λ*x*. *if IEEE.is_nan x* ∨ *IEEE.is_infinity x then undefined*
else integer_of_int ⌊*valof* (*intround float_To_zero* (*valof x*) :: (*11*, *52*) *float*)⌋
⟨*proof*⟩

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

1.4 Linear ordering

definition *lcompare_double* :: *double* ⇒ *double* ⇒ *integer* **where**
lcompare_double x y = (*if is_zero x* ∧ *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* ⟷ *fcompare_double y x* = *ccode.Lt*
⟨*proof*⟩

lemma *fcompare_double_refl*: ¬ *is_nan x* ⟹ *fcompare_double x x* = *ccode.Eq*
⟨*proof*⟩

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

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

lemma *fcompare_double_Lt_trans*: *fcompare_double x y* = *ccode.Lt* ⟹ *fcompare_double y z* = *ccode.Lt*

$\Longrightarrow$ *fcompare_double x z = ccode.Lt*
⟨*proof*⟩

lemma *fcompare_double_eq*: ¬ *is_zero x* ∨ ¬ *is_zero y* $\Longrightarrow$ *fcompare_double x y = ccode.Eq* $\Longrightarrow$ *x = y*
⟨*proof*⟩

lemma *fcompare_double_Lt_asym*: *fcompare_double x y = ccode.Lt* $\Longrightarrow$ *fcompare_double y x = ccode.Lt* $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *compare_double_swap*: *0 < compare_double x y* $\longleftrightarrow$ *compare_double y x < 0*
⟨*proof*⟩

lemma *compare_double_refl*: *compare_double x x = 0*
⟨*proof*⟩

lemma *compare_double_trans*: *compare_double x y* ≤ *0* $\Longrightarrow$ *compare_double y z* ≤ *0* $\Longrightarrow$ *compare_double x z* ≤ *0*
⟨*proof*⟩

lemma *compare_double_antisym*: *compare_double x y* ≤ *0* $\Longrightarrow$ *compare_double y x* ≤ *0* $\Longrightarrow$
¬ *is_zero x* ∨ ¬ *is_zero y* $\Longrightarrow$ *x = y*
⟨*proof*⟩

lemma *zero_compare_double_copysign*: *compare_double* (*copysign_double 1 x*) (*copysign_double 1 y*) ≤ *0* $\Longrightarrow$
is_zero x $\Longrightarrow$ *is_zero y* $\Longrightarrow$ *compare_double x y* ≤ *0*
⟨*proof*⟩

lemma *is_zero_double_cases*: *is_zero x* $\Longrightarrow$ (*x = 0* $\Longrightarrow$ *P*) $\Longrightarrow$ (*x = −0* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *copysign_1_0*[*simp*]: *copysign_double 1 0 = 1 copysign_double 1* (*−0*) *= −1*
⟨*proof*⟩

lemma *is_zero_uminus_double*[*simp*]: *is_zero* (*− x*) $\longleftrightarrow$ *is_zero x*
⟨*proof*⟩

lemma *not_is_zero_one_double*[*simp*]: ¬ *is_zero 1*
⟨*proof*⟩

lemma *uminus_one_neq_one_double*[*simp*]: *− 1* ≠ (*1* :: *double*)
⟨*proof*⟩

definition *lle_double* :: *double* ⇒ *double* ⇒ *bool* **where**
lle_double x y $\longleftrightarrow$ *lcompare_double x y* ≤ *0*

definition *lless_double* :: *double* ⇒ *double* ⇒ *bool* **where**
lless_double x y $\longleftrightarrow$ *lcompare_double x y < 0*

lemma *lcompare_double_ge_0*: *lcompare_double x y* ≥ *0* $\longleftrightarrow$ *lle_double y x*
⟨*proof*⟩

lemma *lcompare_double_gt_0*: *lcompare_double x y > 0* $\longleftrightarrow$ *lless_double y x*
⟨*proof*⟩

lemma *lcompare_double_eq_0*: *lcompare_double x y = 0* $\longleftrightarrow$ *x = y*
⟨*proof*⟩

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

interpretation *double_linorder*: *linorder lle_double lless_double*
⟨*proof*⟩

instantiation *double* :: *equal*
begin

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

instance ⟨*proof*⟩

end

derive (*eq*) *ceq double*

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

lemma *comparator_double*: *comparator comparator_double*
⟨*proof*⟩

⟨*ML*⟩

derive *ccompare double*

1.4.1 Code setup

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

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

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

code_reserved *OCaml Pervasives FloatUtil*

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

hide_const (**open**) *fcompare_double*

2 Event parameters

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

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

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

datatype *event_data* = *EInt integer* | *EFloat double* | *EString String.literal*

derive (*eq*) *ceq event_data*
derive *ccompare event_data*

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

fun *less_eq_event_data* **where**
  EInt x ≤ *EInt y* ⟷ *x* ≤ *y*
| *EInt x* ≤ *EFloat y* ⟷ *double_of_integer x* ≤ *y*
| *EInt _* ≤ *EString _* ⟷ *False*
| *EFloat x* ≤ *EInt y* ⟷ *x* ≤ *double_of_integer y*
| *EFloat x* ≤ *EFloat y* ⟷ *x* ≤ *y*

```
| EFloat _ ≤ EString _ ⟷ False
| EString x ≤ EString y ⟷ lexordp_eq (String.explode x) (String.explode y)
| EString _ ≤ _ ⟷ False

definition less_event_data :: event_data ⇒ 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 ⟨proof⟩

end

primrec integer_of_event_data :: event_data ⇒ 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 ⇒ double where
  double_of_event_data (EInt x) = double_of_integer x
| double_of_event_data (EFloat x) = x
| double_of_event_data (EString _) = nan
```

3 Regular expressions

context begin

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

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

definition *Wild* = *Skip 1*

lemma *size_regex_estimation*[*termination_simp*]: $x \in atms\ r \Longrightarrow y < f\ x \Longrightarrow y < size\_regex\ f\ r$
⟨*proof*⟩

lemma *size_regex_estimation'*[*termination_simp*]: $x \in atms\ r \Longrightarrow y \leq f\ x \Longrightarrow y \leq size\_regex\ f\ r$
⟨*proof*⟩ **definition** *TimesL r S* = *Times r ‘ S*
qualified definition *TimesR R s* = (λ*r*. *Times r s*) ‘ *R*

qualified primrec *fv_regex* **where**
fv_regex fv (*Skip n*) = {}
| *fv_regex fv* (*Test* φ) = *fv* φ
| *fv_regex fv* (*Plus r s*) = *fv_regex fv r* ∪ *fv_regex fv s*
| *fv_regex fv* (*Times r s*) = *fv_regex fv r* ∪ *fv_regex fv s*
| *fv_regex fv* (*Star r*) = *fv_regex fv r*

lemma *fv_regex_cong*[*fundef_cong*]:
$r = r' \Longrightarrow (\bigwedge z.\ z \in atms\ r \Longrightarrow fv\ z = fv'\ z) \Longrightarrow fv\_regex\ fv\ r = fv\_regex\ fv'\ r'$
⟨*proof*⟩

lemma *finite_fv_regex*[*simp*]: $(\bigwedge z.\ z \in atms\ r \Longrightarrow finite\ (fv\ z)) \Longrightarrow finite\ (fv\_regex\ fv\ r)$
⟨*proof*⟩

lemma *fv_regex_commute*:
$(\bigwedge z.\ z \in atms\ r \Longrightarrow x \in fv\ z \longleftrightarrow g\ x \in fv'\ z) \Longrightarrow x \in fv\_regex\ fv\ r \longleftrightarrow g\ x \in fv\_regex\ fv'\ r$
⟨*proof*⟩

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

lemma *insert_Un*: *insert x* (*A* ∪ *B*) = *insert x A* ∪ *insert x B*
⟨*proof*⟩

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

abbreviation *min_regex_default f r j* ≡ (*if atms r* = {} *then j else Min* ((λ*z*. *f z j*) ‘ *atms r*))

qualified primrec *match* :: (*nat* ⇒ *'a* ⇒ *bool*) ⇒ *'a regex* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
match test (*Skip n*) = (λ*i j*. *j* = *i* + *n*)
| *match test* (*Test* φ) = (λ*i j*. *i* = *j* ∧ *test i* φ)

| *match test* (*Plus r s*) = *match test r* ⊔ *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'* ⟹ (⋀*i z*. *z* ∈ *atms r* ⟹ *t i z* = *t' i z*) ⟹ *match t r* = *match t' r'*
⟨*proof*⟩ **primrec** *eps* **where**
eps test i (*Skip n*) = (*n* = *0*)
| *eps test i* (*Test φ*) = *test i φ*
| *eps test i* (*Plus r s*) = (*eps test i r* ∨ *eps test i s*)
| *eps test i* (*Times r s*) = (*eps test i r* ∧ *eps test i s*)
| *eps test i* (*Star r*) = *True*

qualified primrec *lpd* **where**
lpd test i (*Skip n*) = (*case n of 0* ⇒ {} | *Suc m* ⇒ {*Skip m*})
| *lpd test i* (*Test φ*) = {}
| *lpd test i* (*Plus r s*) = (*lpd test i r* ∪ *lpd test i s*)
| *lpd test i* (*Times r s*) = *TimesR* (*lpd test i r*) *s* ∪ (*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 *lpdκ* **where**
lpdκ κ test i (*Skip n*) = (*case n of 0* ⇒ {} | *Suc m* ⇒ {*κ* (*Skip m*)})
| *lpdκ κ test i* (*Test φ*) = {}
| *lpdκ κ test i* (*Plus r s*) = *lpdκ κ test i r* ∪ *lpdκ κ test i s*
| *lpdκ κ test i* (*Times r s*) = *lpdκ* (λ*t*. *κ* (*Times t s*)) *test i r* ∪ (*if eps test i r then lpdκ κ test i s else* {})
| *lpdκ κ test i* (*Star r*) = *lpdκ* (λ*t*. *κ* (*Times t* (*Star r*))) *test i r*

qualified primrec *rpd* **where**
rpd test i (*Skip n*) = (*case n of 0* ⇒ {} | *Suc m* ⇒ {*Skip m*})
| *rpd test i* (*Test φ*) = {}
| *rpd test i* (*Plus r s*) = (*rpd test i r* ∪ *rpd test i s*)
| *rpd test i* (*Times r s*) = *TimesL r* (*rpd test i s*) ∪ (*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 *rpdκ* **where**
rpdκ κ test i (*Skip n*) = (*case n of 0* ⇒ {} | *Suc m* ⇒ {*κ* (*Skip m*)})
| *rpdκ κ test i* (*Test φ*) = {}
| *rpdκ κ test i* (*Plus r s*) = *rpdκ κ test i r* ∪ *rpdκ κ test i s*
| *rpdκ κ test i* (*Times r s*) = *rpdκ* (λ*t*. *κ* (*Times r t*)) *test i s* ∪ (*if eps test i s then rpdκ κ test i r else* {})
| *rpdκ κ test i* (*Star r*) = *rpdκ* (λ*t*. *κ* (*Times* (*Star r*) *t*)) *test i r*

lemma *lpdκ_lpd*: *lpdκ κ test i r* = *κ* ‘ *lpd test i r*
⟨*proof*⟩

lemma *rpdκ_rpd*: *rpdκ κ test i r* = *κ* ‘ *rpd test i r*
⟨*proof*⟩

lemma *match_le*: *match test r i j* ⟹ *i* ≤ *j*
⟨*proof*⟩

lemma *match_rtranclp_le*: (*match test r*)** *i j* ⟹ *i* ≤ *j*
⟨*proof*⟩

lemma *eps_match*: *eps test i r* ⟷ *match test r i i*
⟨*proof*⟩

lemma *lpd_match*: *i* < *j* ⟹ *match test r i j* ⟷ (⨆ *s* ∈ *lpd test i r*. *match test s*) (*i* + *1*) *j*
⟨*proof*⟩

lemma *rpd_match*: $i < j \Longrightarrow match\ test\ r\ i\ j \longleftrightarrow (\bigsqcup s \in rpd\ test\ j\ r.\ match\ test\ s)\ i\ (j - 1)$
⟨*proof*⟩

lemma *lpd_fv_regex*: $s \in lpd\ test\ i\ r \Longrightarrow fv\_regex\ fv\ s \subseteq fv\_regex\ fv\ r$
⟨*proof*⟩

lemma *rpd_fv_regex*: $s \in rpd\ test\ i\ r \Longrightarrow fv\_regex\ fv\ s \subseteq fv\_regex\ fv\ r$
⟨*proof*⟩

lemma *match_fv_cong*:
$(\bigwedge i\ x.\ x \in atms\ r \Longrightarrow test\ i\ x = test'\ i\ x) \Longrightarrow match\ test\ r = match\ test'\ r$
⟨*proof*⟩

lemma *eps_fv_cong*:
$(\bigwedge i\ x.\ x \in atms\ r \Longrightarrow test\ i\ x = test'\ i\ x) \Longrightarrow eps\ test\ i\ r = eps\ test'\ i\ r$
⟨*proof*⟩

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

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

qualified fun *safe_regex* :: *modality* ⇒ *safety* ⇒ $'a$ *regex* ⇒ *bool* **where**
safe_regex m _ (Skip n) = True
| *safe_regex m g (Test φ) = safe g φ*
| *safe_regex m g (Plus r s) = ((g = Lax ∨ fv_regex fv r = fv_regex fv s) ∧ safe_regex m g r ∧ safe_regex m g s)*
| *safe_regex Futu g (Times r s) =*
((g = Lax ∨ fv_regex fv r ⊆ fv_regex fv s) ∧ safe_regex Futu g s ∧ safe_regex Futu Lax r)
| *safe_regex Past g (Times r s) =*
((g = Lax ∨ fv_regex fv s ⊆ fv_regex fv r) ∧ safe_regex Past g r ∧ safe_regex Past Lax s)
| *safe_regex m g (Star r) = ((g = Lax ∨ fv_regex fv r = {}) ∧ safe_regex m g r)*

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

lemma *safe_cosafe*:
$(\bigwedge x.\ x \in atms\ r \Longrightarrow safe\ Strict\ x \Longrightarrow safe\ Lax\ x) \Longrightarrow safe\_regex\ m\ Strict\ r \Longrightarrow safe\_regex\ m\ Lax\ r$
⟨*proof*⟩

lemma *safe_lpd_fv_regex_le*: *safe_regex Futu Strict r* ⟹ $s \in lpd\ test\ i\ r \Longrightarrow fv\_regex\ fv\ r \subseteq fv\_regex\ fv\ s$
⟨*proof*⟩

lemma *safe_lpd_fv_regex*: *safe_regex Futu Strict r* ⟹ $s \in lpd\ test\ i\ r \Longrightarrow fv\_regex\ fv\ s = fv\_regex\ fv\ r$
⟨*proof*⟩

lemma *cosafe_lpd*: *safe_regex Futu Lax r* ⟹ $s \in lpd\ test\ i\ r \Longrightarrow$ *safe_regex Futu Lax s*
⟨*proof*⟩

lemma *safe_lpd*: $(\forall x \in atms\ r.\ safe\ Strict\ x \longrightarrow safe\ Lax\ x) \Longrightarrow$
safe_regex Futu Strict r ⟹ $s \in lpd\ test\ i\ r \Longrightarrow$ *safe_regex Futu Strict s*
⟨*proof*⟩

lemma *safe_rpd_fv_regex_le*: *safe_regex Past Strict r* $\Longrightarrow$ *s* $\in$ *rpd test i r* $\Longrightarrow$ *fv_regex fv r* $\subseteq$ *fv_regex fv s*
⟨*proof*⟩

lemma *safe_rpd_fv_regex*: *safe_regex Past Strict r* $\Longrightarrow$ *s* $\in$ *rpd test i r* $\Longrightarrow$ *fv_regex fv s* = *fv_regex fv r*
⟨*proof*⟩

lemma *cosafe_rpd*: *safe_regex Past Lax r* $\Longrightarrow$ *s* $\in$ *rpd test i r* $\Longrightarrow$ *safe_regex Past Lax s*
⟨*proof*⟩

lemma *safe_rpd*: (∀ *x* $\in$ *atms r*. *safe Strict x* $\longrightarrow$ *safe Lax x*) $\Longrightarrow$
safe_regex Past Strict r $\Longrightarrow$ *s* $\in$ *rpd test i r* $\Longrightarrow$ *safe_regex Past Strict s*
⟨*proof*⟩

lemma *safe_regex_safe*: (⋀*g r*. *safe g r* $\Longrightarrow$ *safe Lax r*) $\Longrightarrow$
safe_regex m g r $\Longrightarrow$ *x* $\in$ *atms r* $\Longrightarrow$ *safe Lax x*
⟨*proof*⟩

lemma *safe_regex_map_regex*:
(⋀*g x*. *x* $\in$ *atms r* $\Longrightarrow$ *safe g x* $\Longrightarrow$ *safe g* (*f x*)) $\Longrightarrow$ (⋀*x*. *x* $\in$ *atms r* $\Longrightarrow$ *fv* (*f x*) = *fv x*) $\Longrightarrow$
safe_regex m g r $\Longrightarrow$ *safe_regex m g* (*map_regex f r*)
⟨*proof*⟩

end

lemma *safe_regex_cong*[*fundef_cong*]:
(⋀*g x*. *x* $\in$ *atms r* $\Longrightarrow$ *safe g x* = *safe*′ *g x*) $\Longrightarrow$
Regex.safe_regex fv safe m g r = *Regex.safe_regex fv safe*′ *m g r*
⟨*proof*⟩

lemma *safe_regex_mono*:
(⋀*g x*. *x* $\in$ *atms r* $\Longrightarrow$ *safe g x* $\Longrightarrow$ *safe*′ *g x*) $\Longrightarrow$
Regex.safe_regex fv safe m g r $\Longrightarrow$ *Regex.safe_regex fv safe*′ *m g r*
⟨*proof*⟩

lemma *match_map_regex*: *match t* (*map_regex f r*) = *match* (λ*k z*. *t k* (*f z*)) *r*
⟨*proof*⟩

lemma *match_cong_strong*:
(⋀*k z*. *k* $\in$ {*i* ..< *j* + *1*} $\Longrightarrow$ *z* $\in$ *atms r* $\Longrightarrow$ *t k z* = *t*′ *k z*) $\Longrightarrow$ *match t r i j* = *match t*′ *r i j*
⟨*proof*⟩

end

4 Metric first-order dynamic logic

derive (*eq*) *ceq enat*

instantiation *enat* :: *ccompare* **begin**
definition *ccompare_enat* :: *enat comparator option* **where**
ccompare_enat = *Some* (λ*x y*. *if x* = *y then order.Eq else if x* < *y then order.Lt else order.Gt*)

instance ⟨*proof*⟩
end

context begin

4.1 Formulas and satisfiability

qualified type_synonym *name* = *String.literal*
qualified type_synonym *event* = (*name* × *event_data list*)
qualified type_synonym *database* = (*name*, *event_data list set list*) *mapping*
qualified type_synonym *prefix* = (*name* × *event_data list*) *prefix*
qualified type_synonym *trace* = (*name* × *event_data list*) *trace*

qualified type_synonym *env* = *event_data list*

4.1.1 Syntax

qualified datatype *trm* = *is_Var*: *Var nat* | *is_Const*: *Const event_data*
| *Plus trm trm* | *Minus trm trm* | *UMinus trm* | *Mult trm trm* | *Div trm trm* | *Mod trm trm*
| *F2i trm* | *I2f trm*

qualified primrec *fvi_trm* :: *nat* ⇒ *trm* ⇒ *nat set* **where**
fvi_trm b (*Var x*) = (*if b* ≤ *x then* {*x* − *b*} *else* {})
| *fvi_trm b* (*Const* _) = {}
| *fvi_trm b* (*Plus x y*) = *fvi_trm b x* ∪ *fvi_trm b y*
| *fvi_trm b* (*Minus x y*) = *fvi_trm b x* ∪ *fvi_trm b y*
| *fvi_trm b* (*UMinus x*) = *fvi_trm b x*
| *fvi_trm b* (*Mult x y*) = *fvi_trm b x* ∪ *fvi_trm b y*
| *fvi_trm b* (*Div x y*) = *fvi_trm b x* ∪ *fvi_trm b y*
| *fvi_trm b* (*Mod x y*) = *fvi_trm b x* ∪ *fvi_trm b y*
| *fvi_trm b* (*F2i x*) = *fvi_trm b x*
| *fvi_trm b* (*I2f x*) = *fvi_trm b x*

abbreviation *fv_trm* ≡ *fvi_trm 0*

qualified primrec *eval_trm* :: *env* ⇒ *trm* ⇒ *event_data* **where**
eval_trm v (*Var x*) = *v* ! *x*
| *eval_trm v* (*Const x*) = *x*
| *eval_trm v* (*Plus x y*) = *eval_trm v x* + *eval_trm v y*
| *eval_trm v* (*Minus x y*) = *eval_trm v x* − *eval_trm v y*
| *eval_trm v* (*UMinus x*) = − *eval_trm v x*
| *eval_trm v* (*Mult x y*) = *eval_trm v x* ∗ *eval_trm v y*
| *eval_trm v* (*Div x y*) = *eval_trm v x div eval_trm v y*
| *eval_trm v* (*Mod x y*) = *eval_trm v x mod eval_trm v y*
| *eval_trm v* (*F2i x*) = *EInt* (*integer_of_event_data* (*eval_trm v x*))
| *eval_trm v* (*I2f x*) = *EFloat* (*double_of_event_data* (*eval_trm v x*))

lemma *eval_trm_fv_cong*: ∀ *x*∈*fv_trm t*. *v* ! *x* = *v*′ ! *x* ⟹ *eval_trm v t* = *eval_trm v*′ *t*
⟨*proof*⟩ **datatype** *agg_type* = *Agg_Cnt* | *Agg_Min* | *Agg_Max* | *Agg_Sum* | *Agg_Avg* | *Agg_Med*
qualified type_synonym *agg_op* = *agg_type* × *event_data*

definition *flatten_multiset* :: (*event_data* × *enat*) *set* ⇒ *event_data list* **where**
flatten_multiset M = *concat* (*map* (λ(*x*, *c*). *replicate* (*the_enat c*) *x*) (*csorted_list_of_set M*))

fun *eval_agg_op* :: *agg_op* ⇒ (*event_data* × *enat*) *set* ⇒ *event_data* **where**
eval_agg_op (*Agg_Cnt*, *y0*) *M* = *EInt* (*integer_of_int* (*length* (*flatten_multiset M*)))
| *eval_agg_op* (*Agg_Min*, *y0*) *M* = (*case flatten_multiset M of*
[] ⇒ *y0*
| *x* # *xs* ⇒ *foldl min x xs*)
| *eval_agg_op* (*Agg_Max*, *y0*) *M* = (*case flatten_multiset M of*
[] ⇒ *y0*
| *x* # *xs* ⇒ *foldl max x xs*)
| *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*

```
    [] ⇒ 0
  | _ ⇒ 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
    if u = 0 then 0 else
      let u′ = u div 2 in
      if even u then
        (double_of_event_data (xs ! (u′−1)) + double_of_event_data (xs ! u′) / double_of_int 2)
      else double_of_event_data (xs ! u′))

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

qualified definition FF = Exists (Neg (Eq (Var 0) (Var 0)))
qualified definition TT ≡ Neg FF

qualified fun fvi :: nat ⇒ formula ⇒ nat set where
  fvi b (Pred r ts) = (⋃t∈set ts. fvi_trm b t)
| fvi b (Let p φ ψ) = fvi b ψ
| fvi b (Eq t1 t2) = fvi_trm b t1 ∪ fvi_trm b t2
| fvi b (Less t1 t2) = fvi_trm b t1 ∪ fvi_trm b t2
| fvi b (LessEq t1 t2) = fvi_trm b t1 ∪ fvi_trm b t2
| fvi b (Neg φ) = fvi b φ
| fvi b (Or φ ψ) = fvi b φ ∪ fvi b ψ
| fvi b (And φ ψ) = fvi b φ ∪ fvi b ψ
| fvi b (Ands φs) = (let xs = map (fvi b) φs in ⋃x∈set xs. x)
| fvi b (Exists φ) = fvi (Suc b) φ
| fvi b (Agg y ω b′ f φ) = fvi (b + b′) φ ∪ fvi_trm (b + b′) f ∪ (if b ≤ 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 ≡ fvi 0
abbreviation fv_regex ≡ Regex.fv_regex fv

lemma fv_abbrevs[simp]: fv TT = {} fv FF = {}
  ⟨proof⟩

lemma fv_subset_Ands: φ ∈ set φs ⟹ fv φ ⊆ fv (Ands φs)
  ⟨proof⟩

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

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

lemma fvi_trm_plus: x ∈ fvi_trm (b + c) t ⟷ x + c ∈ fvi_trm b t
  ⟨proof⟩
```

lemma *fvi_trm_iff_fv_trm*: $x \in fvi\_trm\ b\ t \longleftrightarrow x + b \in fv\_trm\ t$
⟨*proof*⟩

lemma *fvi_plus*: $x \in fvi\ (b + c)\ \varphi \longleftrightarrow x + c \in fvi\ b\ \varphi$
⟨*proof*⟩

lemma *fvi_Suc*: $x \in fvi\ (Suc\ b)\ \varphi \longleftrightarrow Suc\ x \in fvi\ b\ \varphi$
⟨*proof*⟩

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*⟩

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

lemma *fvi_iff_fv*: $x \in fvi\ b\ \varphi \longleftrightarrow x + b \in fv\ \varphi$
⟨*proof*⟩ **definition** *nfv* :: *formula* ⇒ *nat* **where**
$nfv\ \varphi = Max\ (insert\ 0\ (Suc\ `\ fv\ \varphi))$

qualified abbreviation *nfv_regex* **where**
$nfv\_regex \equiv Regex.nfv\_regex\ fv$

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

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

lemma *nfv_Ands*[*simp*]: $nfv\ (Ands\ l) = Max\ (insert\ 0\ (nfv\ `\ set\ l))$
⟨*proof*⟩

lemma *fvi_less_nfv*: $\forall\, i{\in}fv\ \varphi.\ i < nfv\ \varphi$
⟨*proof*⟩

lemma *fvi_less_nfv_regex*: $\forall\, i{\in}fv\_regex\ \varphi.\ i < nfv\_regex\ \varphi$
⟨*proof*⟩

4.1.2 Future reach

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

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

4.1.3 Semantics

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

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

lemma *sat_abbrevs*[*simp*]:
sat σ V v i TT ¬ sat σ V v i FF
⟨*proof*⟩

lemma *sat_Ands*: *sat σ V v i (Ands l) ⟷ (∀φ∈set l. sat σ V v i φ)*
⟨*proof*⟩

lemma *sat_Until_rec*: *sat σ V v i (Until φ I ψ) ⟷*
mem 0 I ∧ sat σ V v i ψ ∨

$(\Delta\ \sigma\ (i + 1) \leq right\ I \wedge sat\ \sigma\ V\ v\ i\ \varphi \wedge sat\ \sigma\ V\ v\ (i + 1)\ (Until\ \varphi\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ \psi))$
(**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

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

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

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

lemma *sat_Since_0*: $sat\ \sigma\ V\ v\ 0\ (Since\ \varphi\ I\ \psi) \longleftrightarrow mem\ 0\ I \wedge sat\ \sigma\ V\ v\ 0\ \psi$
⟨*proof*⟩

lemma *sat_MatchP_0*: $sat\ \sigma\ V\ v\ 0\ (MatchP\ I\ r) \longleftrightarrow mem\ 0\ I \wedge Regex.eps\ (sat\ \sigma\ V\ v)\ 0\ r$
⟨*proof*⟩

lemma *sat_Since_point*: $sat\ \sigma\ V\ v\ i\ (Since\ \varphi\ I\ \psi) \Longrightarrow$
$(\bigwedge j.\ j \leq i \Longrightarrow mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \Longrightarrow sat\ \sigma\ V\ v\ i\ (Since\ \varphi\ (point\ (\tau\ \sigma\ i - \tau\ \sigma\ j))\ \psi) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *sat_MatchP_point*: $sat\ \sigma\ V\ v\ i\ (MatchP\ I\ r) \Longrightarrow$
$(\bigwedge j.\ j \leq i \Longrightarrow mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \Longrightarrow sat\ \sigma\ V\ v\ i\ (MatchP\ (point\ (\tau\ \sigma\ i - \tau\ \sigma\ j))\ r) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *sat_Since_pointD*: $sat\ \sigma\ V\ v\ i\ (Since\ \varphi\ (point\ t)\ \psi) \Longrightarrow mem\ t\ I \Longrightarrow sat\ \sigma\ V\ v\ i\ (Since\ \varphi\ I\ \psi)$
⟨*proof*⟩

lemma *sat_MatchP_pointD*: $sat\ \sigma\ V\ v\ i\ (MatchP\ (point\ t)\ r) \Longrightarrow mem\ t\ I \Longrightarrow sat\ \sigma\ V\ v\ i\ (MatchP\ I\ r)$
⟨*proof*⟩

lemma *sat_fv_cong*: $\forall\, x{\in}fv\ \varphi.\ v!x = v'!x \Longrightarrow sat\ \sigma\ V\ v\ i\ \varphi = sat\ \sigma\ V\ v'\ i\ \varphi$
⟨*proof*⟩

lemma *match_fv_cong*:
$\forall\, x{\in}fv\_regex\ r.\ v!x = v'!x \Longrightarrow Regex.match\ (sat\ \sigma\ V\ v)\ r = Regex.match\ (sat\ \sigma\ V\ v')\ r$
⟨*proof*⟩

lemma *eps_fv_cong*:
$\forall\, x{\in}fv\_regex\ r.\ v!x = v'!x \Longrightarrow Regex.eps\ (sat\ \sigma\ V\ v)\ i\ r = Regex.eps\ (sat\ \sigma\ V\ v')\ i\ r$
⟨*proof*⟩

4.2 Past-only formulas

fun *past_only* :: *formula* ⇒ *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) = (∀ α∈*set* l. *past_only* α)
| *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* < *plen* π ⟹ *dom V* = *dom V*′ ⟹
    (⋀*p i*. *p* ∈ *dom V* ⟹ *i* < *plen* π ⟹ *the* (*V p*) *i* = *the* (*V*′ *p*) *i*) ⟹
    past_only φ ⟹ *sat* σ *V v i* φ = *sat* σ′ *V*′ *v i* φ
⟨*proof*⟩

4.3 Safe formulas

fun *remove_neg* :: *formula* ⇒ *formula* **where**
  remove_neg (*Neg* φ) = φ
| *remove_neg* φ = φ

lemma *fvi_remove_neg*[*simp*]: *fvi b* (*remove_neg* φ) = *fvi b* φ
  ⟨*proof*⟩

lemma *partition_cong*[*fundef_cong*]:
  xs = *ys* ⟹ (⋀*x*. *x*∈*set xs* ⟹ *f x* = *g x*) ⟹ *partition f xs* = *partition g ys*
  ⟨*proof*⟩

lemma *size_remove_neg*[*termination_simp*]: *size* (*remove_neg* φ) ≤ *size* φ
  ⟨*proof*⟩

fun *is_constraint* :: *formula* ⇒ *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* ⇒ *formula* ⇒ *bool* **where**
  safe_assignment X φ = (*case* φ *of*
      Eq (*Var x*) (*Var y*) ⇒ (*x* ∉ *X* ⟷ *y* ∈ *X*)
    | *Eq* (*Var x*) *t* ⇒ (*x* ∉ *X* ∧ *fv_trm t* ⊆ *X*)
    | *Eq t* (*Var x*) ⇒ (*x* ∉ *X* ∧ *fv_trm t* ⊆ *X*)
    | _ ⇒ *False*)

fun *safe_formula* :: *formula* ⇒ *bool* **where**
  safe_formula (*Eq t1 t2*) = (*is_Const t1* ∧ (*is_Const t2* ∨ *is_Var t2*) ∨ *is_Var t1* ∧ *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*) = (∀ *t*∈*set ts*. *is_Var t* ∨ *is_Const t*)
| *safe_formula* (*Let p* φ ψ) = ({*0*..<*nfv* φ} ⊆ *fv* φ ∧ *safe_formula* φ ∧ *safe_formula* ψ)
| *safe_formula* (*Neg* φ) = (*fv* φ = {} ∧ *safe_formula* φ)
| *safe_formula* (*Or* φ ψ) = (*fv* ψ = *fv* φ ∧ *safe_formula* φ ∧ *safe_formula* ψ)
| *safe_formula* (*And* φ ψ) = (*safe_formula* φ ∧
    (*safe_assignment* (*fv* φ) ψ ∨ *safe_formula* ψ ∨
      fv ψ ⊆ *fv* φ ∧ (*is_constraint* ψ ∨ (*case* ψ *of Neg* ψ′ ⇒ *safe_formula* ψ′ | _ ⇒ *False*))))
| *safe_formula* (*Ands l*) = (*let* (*pos*, *neg*) = *partition safe_formula l in pos* ≠ [] ∧
    list_all safe_formula (*map remove_neg neg*) ∧ ⋃(*set* (*map fv neg*)) ⊆ ⋃(*set* (*map fv pos*)))
| *safe_formula* (*Exists* φ) = (*safe_formula* φ)
| *safe_formula* (*Agg y* ω *b f* φ) = (*safe_formula* φ ∧ *y* + *b* ∉ *fv* φ ∧ {*0*..<*b*} ⊆ *fv* φ ∧ *fv_trm f* ⊆ *fv* φ)
| *safe_formula* (*Prev I* φ) = (*safe_formula* φ)
| *safe_formula* (*Next I* φ) = (*safe_formula* φ)
| *safe_formula* (*Since* φ *I* ψ) = (*fv* φ ⊆ *fv* ψ ∧
    (*safe_formula* φ ∨ (*case* φ *of Neg* φ′ ⇒ *safe_formula* φ′ | _ ⇒ *False*)) ∧ *safe_formula* ψ)
| *safe_formula* (*Until* φ *I* ψ) = (*fv* φ ⊆ *fv* ψ ∧
    (*safe_formula* φ ∨ (*case* φ *of Neg* φ′ ⇒ *safe_formula* φ′ | _ ⇒ *False*)) ∧ *safe_formula* ψ)
| *safe_formula* (*MatchP I r*) = *Regex.safe_regex fv* (λ*g* φ. *safe_formula* φ ∨
    (*g* = *Lax* ∧ (*case* φ *of Neg* φ′ ⇒ *safe_formula* φ′ | _ ⇒ *False*))) *Past Strict r*
| *safe_formula* (*MatchF I r*) = *Regex.safe_regex fv* (λ*g* φ. *safe_formula* φ ∨
    (*g* = *Lax* ∧ (*case* φ *of Neg* φ′ ⇒ *safe_formula* φ′ | _ ⇒ *False*))) *Futu Strict r*

abbreviation *safe_regex* ≡ *Regex.safe_regex fv* (λ*g* φ. *safe_formula* φ ∨
  (*g* = *Lax* ∧ (*case* φ *of Neg* φ′ ⇒ *safe_formula* φ′ | _ ⇒ *False*)))

lemma *safe_regex_safe_formula*:
  safe_regex m g r ⟹ φ ∈ *Regex.atms r* ⟹ *safe_formula* φ ∨
  (∃ψ. φ = *Neg* ψ ∧ *safe_formula* ψ)
  ⟨*proof*⟩

lemma *safe_abbrevs*[*simp*]: *safe_formula TT safe_formula FF*
  ⟨*proof*⟩

definition *safe_neg* :: *formula* ⇒ *bool* **where**
  safe_neg φ ⟷ (¬ *safe_formula* φ ⟶ *safe_formula* (*remove_neg* φ))

definition *atms* :: *formula Regex.regex* ⇒ *formula set* **where**
  atms r = (⋃φ ∈ *Regex.atms r*.
    if safe_formula φ *then* {φ} *else case* φ *of Neg* φ′ ⇒ {φ′} | _ ⇒ {})

lemma *atms_simps*[*simp*]:
  atms (*Regex.Skip n*) = {}
  atms (*Regex.Test* φ) = (*if safe_formula* φ *then* {φ} *else case* φ *of Neg* φ′ ⇒ {φ′} | _ ⇒ {})
  atms (*Regex.Plus r s*) = *atms r* ∪ *atms s*
  atms (*Regex.Times r s*) = *atms r* ∪ *atms s*
  atms (*Regex.Star r*) = *atms r*
  ⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *safe_formula_induct*[*consumes 1*, *case_names Eq_Const Eq_Var1 Eq_Var2 neq_Var Pred Let And_assign And_safe And_constraint And_Not Ands Neg Or Exists Agg Prev Next Since Not_Since Until Not_Until MatchP MatchF*]:
assumes *safe_formula* φ
and *Eq_Const*: $\bigwedge c\ d.\ P\ (Eq\ (Const\ c)\ (Const\ d))$
and *Eq_Var1*: $\bigwedge c\ x.\ P\ (Eq\ (Const\ c)\ (Var\ x))$
and *Eq_Var2*: $\bigwedge c\ x.\ P\ (Eq\ (Var\ x)\ (Const\ c))$
and *neq_Var*: $\bigwedge x.\ P\ (Neg\ (Eq\ (Var\ x)\ (Var\ x)))$
and *Pred*: $\bigwedge e\ ts.\ \forall\, t{\in}set\ ts.\ is\_Var\ t \vee is\_Const\ t \Longrightarrow P\ (Pred\ e\ ts)$
and *Let*: $\bigwedge p\ \varphi\ \psi.\ \{0..{<}nfv\ \varphi\} \subseteq fv\ \varphi \Longrightarrow safe\_formula\ \varphi \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Let\ p\ \varphi\ \psi)$
and *And_assign*: $\bigwedge \varphi\ \psi.\ safe\_formula\ \varphi \Longrightarrow safe\_assignment\ (fv\ \varphi)\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ (And\ \varphi\ \psi)$
and *And_safe*: $\bigwedge \varphi\ \psi.\ safe\_formula\ \varphi \Longrightarrow \neg\ safe\_assignment\ (fv\ \varphi)\ \psi \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (And\ \varphi\ \psi)$
and *And_constraint*: $\bigwedge \varphi\ \psi.\ safe\_formula\ \varphi \Longrightarrow \neg\ safe\_assignment\ (fv\ \varphi)\ \psi \Longrightarrow \neg\ safe\_formula\ \psi \Longrightarrow fv\ \psi \subseteq fv\ \varphi \Longrightarrow is\_constraint\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ (And\ \varphi\ \psi)$
and *And_Not*: $\bigwedge \varphi\ \psi.\ safe\_formula\ \varphi \Longrightarrow \neg\ safe\_assignment\ (fv\ \varphi)\ (Neg\ \psi) \Longrightarrow \neg\ safe\_formula\ (Neg\ \psi) \Longrightarrow fv\ (Neg\ \psi) \subseteq fv\ \varphi \Longrightarrow \neg\ is\_constraint\ (Neg\ \psi) \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (And\ \varphi\ (Neg\ \psi))$
and *Ands*: $\bigwedge l\ pos\ neg.\ (pos, neg) = partition\ safe\_formula\ l \Longrightarrow pos \neq [] \Longrightarrow list\_all\ safe\_formula\ pos \Longrightarrow list\_all\ safe\_formula\ (map\ remove\_neg\ neg) \Longrightarrow (\bigcup \varphi{\in}set\ neg.\ fv\ \varphi) \subseteq (\bigcup \varphi{\in}set\ pos.\ fv\ \varphi) \Longrightarrow list\_all\ P\ pos \Longrightarrow list\_all\ P\ (map\ remove\_neg\ neg) \Longrightarrow P\ (Ands\ l)$
and *Neg*: $\bigwedge \varphi.\ fv\ \varphi = \{\} \Longrightarrow safe\_formula\ \varphi \Longrightarrow P\ \varphi \Longrightarrow P\ (Neg\ \varphi)$
and *Or*: $\bigwedge \varphi\ \psi.\ fv\ \psi = fv\ \varphi \Longrightarrow safe\_formula\ \varphi \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Or\ \varphi\ \psi)$
and *Exists*: $\bigwedge \varphi.\ safe\_formula\ \varphi \Longrightarrow P\ \varphi \Longrightarrow P\ (Exists\ \varphi)$
and *Agg*: $\bigwedge y\ \omega\ b\ f\ \varphi.\ y + b \notin fv\ \varphi \Longrightarrow \{0..{<}b\} \subseteq fv\ \varphi \Longrightarrow fv\_trm\ f \subseteq fv\ \varphi \Longrightarrow safe\_formula\ \varphi \Longrightarrow P\ \varphi \Longrightarrow P\ (Agg\ y\ \omega\ b\ f\ \varphi)$
and *Prev*: $\bigwedge I\ \varphi.\ safe\_formula\ \varphi \Longrightarrow P\ \varphi \Longrightarrow P\ (Prev\ I\ \varphi)$
and *Next*: $\bigwedge I\ \varphi.\ safe\_formula\ \varphi \Longrightarrow P\ \varphi \Longrightarrow P\ (Next\ I\ \varphi)$
and *Since*: $\bigwedge \varphi\ I\ \psi.\ fv\ \varphi \subseteq fv\ \psi \Longrightarrow safe\_formula\ \varphi \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Since\ \varphi\ I\ \psi)$
and *Not_Since*: $\bigwedge \varphi\ I\ \psi.\ fv\ (Neg\ \varphi) \subseteq fv\ \psi \Longrightarrow safe\_formula\ \varphi \Longrightarrow \neg\ safe\_formula\ (Neg\ \varphi) \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Since\ (Neg\ \varphi)\ I\ \psi\ )$
and *Until*: $\bigwedge \varphi\ I\ \psi.\ fv\ \varphi \subseteq fv\ \psi \Longrightarrow safe\_formula\ \varphi \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Until\ \varphi\ I\ \psi)$
and *Not_Until*: $\bigwedge \varphi\ I\ \psi.\ fv\ (Neg\ \varphi) \subseteq fv\ \psi \Longrightarrow safe\_formula\ \varphi \Longrightarrow \neg\ safe\_formula\ (Neg\ \varphi) \Longrightarrow safe\_formula\ \psi \Longrightarrow P\ \varphi \Longrightarrow P\ \psi \Longrightarrow P\ (Until\ (Neg\ \varphi)\ I\ \psi)$
and *MatchP*: $\bigwedge I\ r.\ safe\_regex\ Past\ Strict\ r \Longrightarrow \forall\, \varphi \in atms\ r.\ P\ \varphi \Longrightarrow P\ (MatchP\ I\ r)$
and *MatchF*: $\bigwedge I\ r.\ safe\_regex\ Futu\ Strict\ r \Longrightarrow \forall\, \varphi \in atms\ r.\ P\ \varphi \Longrightarrow P\ (MatchF\ I\ r)$
shows $P\ \varphi$
⟨*proof*⟩

lemma *safe_formula_NegD*:
$safe\_formula\ (Formula.Neg\ \varphi) \Longrightarrow fv\ \varphi = \{\} \vee (\exists\, x.\ \varphi = Formula.Eq\ (Formula.Var\ x)\ (Formula.Var\ x))$
⟨*proof*⟩

4.4 Slicing traces

qualified fun *matches* ::
$env \Rightarrow formula \Rightarrow name \times event\_data\ list \Rightarrow bool$ **where**
$matches\ v\ (Pred\ r\ ts)\ e = (fst\ e = r \wedge map\ (eval\_trm\ v)\ ts = snd\ e)$

| *matches v (Let p φ ψ) e =*
((∃ v'. matches v' φ e ∧ matches v ψ (p, v')) ∨
fst e ≠ p ∧ matches v ψ e)
| *matches v (Eq _ _) e = False*
| *matches v (Less _ _) e = False*
| *matches v (LessEq _ _) e = False*
| *matches v (Neg φ) e = matches v φ e*
| *matches v (Or φ ψ) e = (matches v φ e ∨ matches v ψ e)*
| *matches v (And φ ψ) e = (matches v φ e ∨ matches v ψ e)*
| *matches v (Ands l) e = (∃ φ∈set l. matches v φ e)*
| *matches v (Exists φ) e = (∃ z. matches (z # v) φ e)*
| *matches v (Agg y ω b f φ) e = (∃ zs. length zs = b ∧ matches (zs @ v) φ e)*
| *matches v (Prev I φ) e = matches v φ e*
| *matches v (Next I φ) e = matches v φ e*
| *matches v (Since φ I ψ) e = (matches v φ e ∨ matches v ψ e)*
| *matches v (Until φ I ψ) e = (matches v φ e ∨ matches v ψ e)*
| *matches v (MatchP I r) e = (∃ φ ∈ Regex.atms r. matches v φ e)*
| *matches v (MatchF I r) e = (∃ φ ∈ Regex.atms r. matches v φ e)*

lemma *matches_cong*:
∀ x∈fv φ. v!x = v'!x ⟹ matches v φ e = matches v' φ e
⟨*proof*⟩

abbreviation *relevant_events* **where** *relevant_events φ S ≡ {e. S ∩ {v. matches v φ e} ≠ {}}*

lemma *sat_slice_strong*:
assumes *v ∈ S dom V = dom V'*
⋀p v i. p ∈ dom V ⟹ (p, v) ∈ relevant_events φ S ⟹ v ∈ the (V p) i ⟷ v ∈ the (V' p) i
shows *relevant_events φ S − {e. fst e ∈ dom V} ⊆ E ⟹*
sat σ V v i φ ⟷ sat (map_Γ (λD. D ∩ E) σ) V' v i φ
⟨*proof*⟩

4.5 Translation to n-ary conjunction

fun *get_and_list* :: *formula ⇒ formula list* **where**
get_and_list (Ands l) = l
| *get_and_list φ = [φ]*

lemma *fv_get_and*: *(⋃x∈(set (get_and_list φ)). fvi b x) = fvi b φ*
⟨*proof*⟩

lemma *safe_get_and*: *safe_formula φ ⟹ list_all safe_neg (get_and_list φ)*
⟨*proof*⟩

lemma *sat_get_and*: *sat σ V v i φ ⟷ list_all (sat σ V v i) (get_and_list φ)*
⟨*proof*⟩

fun *convert_multiway* :: *formula ⇒ formula* **where**
convert_multiway (Neg φ) = Neg (convert_multiway φ)
| *convert_multiway (Or φ ψ) = Or (convert_multiway φ) (convert_multiway ψ)*
| *convert_multiway (And φ ψ) = (if safe_assignment (fv φ) ψ then*
And (convert_multiway φ) ψ
else if safe_formula ψ then
Ands (get_and_list (convert_multiway φ) @ get_and_list (convert_multiway ψ))
else if is_constraint ψ then
And (convert_multiway φ) ψ
else Ands (convert_multiway ψ # get_and_list (convert_multiway φ)))
| *convert_multiway (Exists φ) = Exists (convert_multiway φ)*

| *convert_multiway* (*Agg y ω b f φ*) = *Agg y ω b f* (*convert_multiway φ*)
| *convert_multiway* (*Prev I φ*) = *Prev I* (*convert_multiway φ*)
| *convert_multiway* (*Next I φ*) = *Next I* (*convert_multiway φ*)
| *convert_multiway* (*Since φ I ψ*) = *Since* (*convert_multiway φ*) *I* (*convert_multiway ψ*)
| *convert_multiway* (*Until φ I ψ*) = *Until* (*convert_multiway φ*) *I* (*convert_multiway ψ*)
| *convert_multiway* (*MatchP I r*) = *MatchP I* (*Regex.map_regex convert_multiway r*)
| *convert_multiway* (*MatchF I r*) = *MatchF I* (*Regex.map_regex convert_multiway r*)
| *convert_multiway φ* = *φ*

abbreviation *convert_multiway_regex* ≡ *Regex.map_regex convert_multiway*

lemma *fv_safe_get_and*:
safe_formula φ ⟹ *fv φ* ⊆ (⋃ *x*∈(*set* (*filter safe_formula* (*get_and_list φ*))). *fv x*)
⟨*proof*⟩

lemma *ex_safe_get_and*:
safe_formula φ ⟹ *list_ex safe_formula* (*get_and_list φ*)
⟨*proof*⟩

lemma *case_NegE*: (*case φ of Neg φ′* ⇒ *P φ′* | _ ⇒ *False*) ⟹ (⋀*φ′*. *φ* = *Neg φ′* ⟹ *P φ′* ⟹ *Q*)
⟹ *Q*
⟨*proof*⟩

lemma *convert_multiway_remove_neg*: *safe_formula* (*remove_neg φ*) ⟹ *convert_multiway* (*remove_neg φ*) = *remove_neg* (*convert_multiway φ*)
⟨*proof*⟩

lemma *fv_convert_multiway*: *safe_formula φ* ⟹ *fvi b* (*convert_multiway φ*) = *fvi b φ*
⟨*proof*⟩

lemma *get_and_nonempty*:
assumes *safe_formula φ*
shows *get_and_list φ* ≠ []
⟨*proof*⟩

lemma *future_bounded_get_and*:
list_all future_bounded (*get_and_list φ*) = *future_bounded φ*
⟨*proof*⟩

lemma *safe_convert_multiway*: *safe_formula φ* ⟹ *safe_formula* (*convert_multiway φ*)
⟨*proof*⟩

lemma *future_bounded_convert_multiway*: *safe_formula φ* ⟹ *future_bounded* (*convert_multiway φ*) = *future_bounded φ*
⟨*proof*⟩

lemma *sat_convert_multiway*: *safe_formula φ* ⟹ *sat σ V v i* (*convert_multiway φ*) ⟷ *sat σ V v i φ*
⟨*proof*⟩

end

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

lemma *sat_slice_iff*:
assumes *v* ∈ *S*
shows *Formula.sat σ V v i φ* ⟷ *Formula.sat* (*Formula_slicer.slice φ S σ*) *V v i φ*
⟨*proof*⟩

lemma *Neg_splits*:
P (*case* φ *of formula.Neg* $\psi \Rightarrow f\ \psi \mid \varphi \Rightarrow g\ \varphi$) =
(($\forall \psi.\ \varphi =$ *formula.Neg* $\psi \longrightarrow P\ (f\ \psi)) \wedge ((\neg$ *Formula.is_Neg* $\varphi) \longrightarrow P\ (g\ \varphi)))$
P (*case* φ *of formula.Neg* $\psi \Rightarrow f\ \psi \mid \_ \Rightarrow g\ \varphi$) =
$(\neg ((\exists \psi.\ \varphi =$ *formula.Neg* $\psi \wedge \neg\ P\ (f\ \psi)) \vee ((\neg$ *Formula.is_Neg* $\varphi) \wedge \neg\ P\ (g\ \varphi))))$
⟨*proof*⟩

5 Optimized relational join

5.1 Binary join

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

fun *proj_tuple* :: *bool list* ⇒ *'a tuple* ⇒ *'a 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*: $(\bigwedge i.\ i \in$ *set bs* $\Longrightarrow \neg i) \Longrightarrow$ *length bs* = *length as* ⟹
proj_tuple bs as = *replicate* (*length bs*) *None*
⟨*proof*⟩

lemma *proj_tuple_join_mask_empty*: *length as* = *n* ⟹
proj_tuple (*join_mask n* {}) *as* = *replicate n None*
⟨*proof*⟩

lemma *proj_tuple_alt*: *proj_tuple bs as* = *map2* ($\lambda b\ a.$ *if b then a else None*) *bs as*
⟨*proof*⟩

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

lemma *proj_tuple_join_mask_restrict*: *length as* = *n* ⟹
proj_tuple (*join_mask n X*) *as* = *restrict X as*
⟨*proof*⟩

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

lemma *wf_tuple_change_base*:
assumes *wf*: *wf_tuple n X as*
and *mask*: *join_mask n X* = *join_mask n Y*
shows *wf_tuple n Y as*
⟨*proof*⟩

definition *proj_tuple_in_join* :: *bool* ⇒ *bool list* ⇒ *'a tuple* ⇒ *'a table* ⇒ *bool* **where**
proj_tuple_in_join pos bs as t = (*if pos then proj_tuple bs as* ∈ *t else proj_tuple bs as* ∉ *t*)

abbreviation *join_cond pos t* ≡ ($\lambda as.$ *if pos then as* ∈ *t else as* ∉ *t*)

abbreviation *join_filter_cond pos t* ≡ ($\lambda as\ \_.$ *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* ⟷ *join_cond pos t as*
⟨*proof*⟩

lemma *join_sub*:
assumes *L* ⊆ *R table n L t1 table n R t2*
shows *join t2 pos t1* = {*as* ∈ *t2*. *proj_tuple_in_join pos* (*join_mask n L*) *as t1*}
⟨*proof*⟩

lemma *join_sub'*:
assumes *R* ⊆ *L table n L t1 table n R t2*
shows *join t2 True t1* = {*as* ∈ *t1*. *proj_tuple_in_join True* (*join_mask n R*) *as t2*}
⟨*proof*⟩

lemma *join_eq*:
assumes *tab*: *table n R t1 table n R t2*
shows *join t2 pos t1* = (*if pos then t2* ∩ *t1 else t2* − *t1*)
⟨*proof*⟩

lemma *join_no_cols*:
assumes *tab*: *table n* {} *t1 table n R t2*
shows *join t2 pos t1* = (*if* (*pos* ⟷ *replicate n None* ∈ *t1*) *then t2 else* {})
⟨*proof*⟩

lemma *join_empty_left*: *join* {} *pos t* = {}
⟨*proof*⟩

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

fun *bin_join* :: *nat* ⇒ *nat set* ⇒ *'a table* ⇒ *bool* ⇒ *nat set* ⇒ *'a table* ⇒ *'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* ⟷ *replicate n None* ∈ *t'*) *then t else* {})
else if A' = *A then* (*if pos then t* ∩ *t' else t* − *t'*)
else if A' ⊆ *A then* {*as* ∈ *t*. *proj_tuple_in_join pos* (*join_mask n A'*) *as t'*}
else if A ⊆ *A'* ∧ *pos then* {*as* ∈ *t'*. *proj_tuple_in_join pos* (*join_mask n A*) *as t*}
else join t pos t')

lemma *bin_join_table*:
assumes *tab*: *table n A t table n A' t'*
shows *bin_join n A t pos A' t'* = *join t pos t'*
⟨*proof*⟩

5.2 Multi-way join

fun *mmulti_join'* :: (*nat set list* ⇒ *nat set list* ⇒ *'a table list* ⇒ *'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* ≠ []
and *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*) (*A_pos* @ *A_neg*) *L*
shows *z* ∈ *mmulti_join' A_pos A_neg L* ⟷ *wf_tuple n* (⋃ *A*∈*set A_pos*. *A*) *z* ∧

list_all2 (λ*A X*. *restrict A z* ∈ *X*) *A_pos* (*take* (*length A_pos*) *L*) ∧
list_all2 (λ*A X*. *restrict A z* ∉ *X*) *A_neg* (*drop* (*length A_pos*) *L*)
⟨*proof*⟩

lemmas *restrict_nested* = *New_max.restrict_nested*

lemma *list_all2_opt_True*:
assumes *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*) ((*A_zs* @ *A_x* # *A_xs* @ *A_y* # *A_ys*) @ *A_neg*)
((*zs* @ *x* # *xs* @ *y* # *ys*) @ *L_neg*)
length A_xs = *length xs length A_ys* = *length ys length A_zs* = *length zs*
shows *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*)
((*A_zs* @ (*A_x* ∪ *A_y*) # *A_xs* @ *A_ys*) @ *A_neg*) ((*zs* @ *join x True y* # *xs* @ *ys*) @ *L_neg*)
⟨*proof*⟩

lemma *mmulti_join'_opt_True*:
assumes *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*) ((*A_zs* @ *A_x* # *A_xs* @ *A_y* # *A_ys*) @ *A_neg*)
((*zs* @ *x* # *xs* @ *y* # *ys*) @ *L_neg*)
length A_xs = *length xs length A_ys* = *length ys length A_zs* = *length zs*
shows *mmulti_join'* (*A_zs* @ *A_x* # *A_xs* @ *A_y* # *A_ys*) *A_neg* ((*zs* @ *x* # *xs* @ *y* # *ys*) @ *L_neg*) =
mmulti_join' (*A_zs* @ (*A_x* ∪ *A_y*) # *A_xs* @ *A_ys*) *A_neg*
((*zs* @ *join x True y* # *xs* @ *ys*) @ *L_neg*)
⟨*proof*⟩

lemma *list_all2_opt_False*:
assumes *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*)
((*A_zs* @ *A_x* # *A_xs*) @ (*A_ws* @ *A_y* # *A_ys*)) ((*zs* @ *x* # *xs*) @ (*ws* @ *y* # *ys*))
length A_ws = *length ws length A_xs* = *length xs*
length A_ys = *length ys length A_zs* = *length zs*
A_y ⊆ *A_x*
shows *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*)
((*A_zs* @ *A_x* # *A_xs*) @ (*A_ws* @ *A_ys*)) ((*zs* @ *join x False y* # *xs*) @ (*ws* @ *ys*))
⟨*proof*⟩

lemma *mmulti_join'_opt_False*:
assumes *list_all2* (λ*A X*. *table n A X* ∧ *wf_set n A*)
((*A_zs* @ *A_x* # *A_xs*) @ (*A_ws* @ *A_y* # *A_ys*)) ((*zs* @ *x* # *xs*) @ (*ws* @ *y* # *ys*))
length A_ws = *length ws length A_xs* = *length xs*
length A_ys = *length ys length A_zs* = *length zs*
A_y ⊆ *A_x*
shows *mmulti_join'* (*A_zs* @ *A_x* # *A_xs*) (*A_ws* @ *A_y* # *A_ys*) ((*zs* @ *x* # *xs*) @ (*ws* @ *y* # *ys*)) =
mmulti_join' (*A_zs* @ *A_x* # *A_xs*) (*A_ws* @ *A_ys*) ((*zs* @ *join x False y* # *xs*) @ (*ws* @ *ys*))
⟨*proof*⟩

fun *find_sub_in* :: *'a set* ⇒ *'a set list* ⇒ *bool* ⇒
(*'a set list* × *'a set* × *'a set list*) *option* **where**
find_sub_in X [] *b* = *None*
| *find_sub_in X* (*x* # *xs*) *b* = (*if* (*x* ⊆ *X* ∨ (*b* ∧ *X* ⊆ *x*)) *then Some* ([], *x*, *xs*)
else (*case find_sub_in X xs b of None* ⇒ *None* | *Some* (*ys*, *z*, *zs*) ⇒ *Some* (*x* # *ys*, *z*, *zs*)))

lemma *find_sub_in_sound*: *find_sub_in X xs b* = *Some* (*ys*, *z*, *zs*) ⟹
xs = *ys* @ *z* # *zs* ∧ (*z* ⊆ *X* ∨ (*b* ∧ *X* ⊆ *z*))
⟨*proof*⟩

fun *find_sub_True* :: *'a set list* ⇒

('a set list × 'a set × 'a set list × 'a set × 'a set list) option **where**
find_sub_True [] = *None*
| *find_sub_True* (x # xs) = (*case find_sub_in x xs True of None* ⇒
(*case find_sub_True xs of None* ⇒ *None*
| *Some* (ys, w, ws, z, zs) ⇒ *Some* (x # ys, w, ws, z, zs))
| *Some* (ys, z, zs) ⇒ *Some* ([], x, ys, z, zs))

lemma *find_sub_True_sound*: *find_sub_True xs* = *Some* (ys, w, ws, z, zs) ⟹
xs = *ys* @ *w* # *ws* @ *z* # *zs* ∧ (*z* ⊆ *w* ∨ *w* ⊆ *z*)
⟨*proof*⟩

fun *find_sub_False* :: 'a set list ⇒ 'a set list ⇒
(('a set list × 'a set × 'a set list) × ('a set list × 'a set × 'a set list)) option **where**
find_sub_False [] *ns* = *None*
| *find_sub_False* (x # xs) *ns* = (*case find_sub_in x ns False of None* ⇒
(*case find_sub_False xs ns of None* ⇒ *None*
| *Some* ((rs, w, ws), (ys, z, zs)) ⇒ *Some* ((x # rs, w, ws), (ys, z, zs)))
| *Some* (ys, z, zs) ⇒ *Some* (([], x, xs), (ys, z, zs)))

lemma *find_sub_False_sound*: *find_sub_False xs ns* = *Some* ((rs, w, ws), (ys, z, zs)) ⟹
xs = *rs* @ *w* # *ws* ∧ *ns* = *ys* @ *z* # *zs* ∧ (*z* ⊆ *w*)
⟨*proof*⟩

fun *proj_list_3* :: 'a list ⇒ ('b list × 'b × 'b list) ⇒ ('a list × 'a × '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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

fun *proj_list_5* :: 'a list ⇒
('b list × 'b × 'b list × 'b × 'b list) ⇒
('a list × 'a × 'a list × 'a × '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*⟩

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'*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *dominate_False_sound*:
assumes *dominate_False A_pos L_pos A_neg L_neg* =
Some (((*A_zs*, *A_x*, *A_xs*), *A_ws*, *A_y*, *A_ys*), ((*zs*, *x*, *xs*), *ws*, *y*, *ys*))
length A_pos = *length L_pos length A_neg* = *length L_neg*
shows *A_pos* = (*A_zs* @ *A_x* # *A_xs*) *A_neg* = *A_ws* @ *A_y* # *A_ys*
L_pos = (*zs* @ *x* # *xs*) *L_neg* = *ws* @ *y* # *ys*
length A_ws = *length ws length A_xs* = *length xs*
length A_ys = *length ys length A_zs* = *length zs*
A_y ⊆ *A_x*
⟨*proof*⟩

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*)))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *mmulti_join_link*:
assumes *A_pos* ≠ []
and *list_all2* (λ*A 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*
⟨*proof*⟩

lemma *mmulti_join_correct*:
assumes *A_pos* ≠ []
and *list_all2* (λ*A 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 (λ*A X*. *restrict A z* ∈ *X*) *A_pos* (*take* (*length A_pos*) *L*) ∧
list_all2 (λ*A X*. *restrict A z* ∉ *X*) *A_neg* (*drop* (*length A_pos*) *L*)
⟨*proof*⟩

6 Generic monitoring algorithm

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

6.1 Monitorable formulas

definition *mmonitorable* φ ⟷ *safe_formula* φ ∧ *Formula.future_bounded* φ
definition *mmonitorable_regex b g r* ⟷ *safe_regex b g r* ∧ *Regex.pred_regex Formula.future_bounded r*

definition *is_simple_eq* :: *Formula.trm* ⇒ *Formula.trm* ⇒ *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* :: *Formula.formula* ⇒ *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* (λ*t*. *Formula.is_Var t* ∨ *Formula.is_Const t*) *ts*
| *mmonitorable_exec* (*Formula.Let p φ ψ*) = ({*0*..<*Formula.nfv φ*} ⊆ *Formula.fv φ* ∧ *mmonitorable_exec φ* ∧ *mmonitorable_exec ψ*)
| *mmonitorable_exec* (*Formula.Neg φ*) = (*fv φ* = {} ∧ *mmonitorable_exec φ*)
| *mmonitorable_exec* (*Formula.Or φ ψ*) = (*fv φ* = *fv ψ* ∧ *mmonitorable_exec φ* ∧ *mmonitorable_exec ψ*)
| *mmonitorable_exec* (*Formula.And φ ψ*) = (*mmonitorable_exec φ* ∧
(*safe_assignment* (*fv φ*) *ψ* ∨ *mmonitorable_exec ψ* ∨
fv ψ ⊆ *fv φ* ∧ (*is_constraint ψ* ∨ (*case ψ of Formula.Neg ψ'* ⇒ *mmonitorable_exec ψ'* | _ ⇒ *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* (λ*g 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* (λ*g 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*)
⟨*proof*⟩

lemma *safe_formula_mmonitorable_exec*: *safe_formula φ* ⟹ *Formula.future_bounded φ* ⟹ *mmonitorable_exec φ*
⟨*proof*⟩

lemma *safe_assignment_future_bounded*: *safe_assignment X φ* ⟹ *Formula.future_bounded φ*
⟨*proof*⟩

lemma *is_constraint_future_bounded*: *is_constraint φ* ⟹ *Formula.future_bounded φ*
⟨*proof*⟩

lemma *mmonitorable_exec_mmonitorable*: *mmonitorable_exec φ* ⟹ *mmonitorable φ*
⟨*proof*⟩

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

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 ∧ ok m s)
| ok m (MTimes r s) = (ok m r ∧ ok m s)
| ok m (MStar r) = ok m r

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

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

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

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

lemma shift_0: shift r 0 = r
  ⟨proof⟩
```

lemma *shift_shift*: *shift* (*shift r k*) *j* = *shift r* (*k* + *j*)
⟨*proof*⟩

lemma *to_mregex_to_mregex_exec*:
case to_mregex r of (*mr*, *φs*) ⇒ *to_mregex_exec r xs* = (*shift mr* (*length xs*), *xs* @ *φs*)
⟨*proof*⟩

lemma *to_mregex_to_mregex_exec_Nil*[*code*]: *to_mregex r* = *to_mregex_exec r* []
⟨*proof*⟩

lemma *ok_mono*: *ok m mr* ⟹ *m* ≤ *n* ⟹ *ok n mr*
⟨*proof*⟩

lemma *from_mregex_cong*: *ok m mr* ⟹ (∀ *i* < *m*. *xs* ! *i* = *ys* ! *i*) ⟹ *from_mregex mr xs* = *from_mregex mr ys*
⟨*proof*⟩

lemma *not_Neg_cases*:
(∀ *ψ*. *φ* ≠ *Formula.Neg ψ*) ⟹ (*case φ of formula.Neg ψ* ⇒ *f ψ* | _ ⇒ *x*) = *x*
⟨*proof*⟩

lemma *to_mregex_exec_ok*:
to_mregex_exec r xs = (*mr*, *ys*) ⟹ ∃ *zs*. *ys* = *xs* @ *zs* ∧ *set zs* = *atms r* ∧ *ok* (*length ys*) *mr*
⟨*proof*⟩

lemma *ok_shift*: *ok* (*i* + *m*) (*Monitor.shift r i*) ⟷ *ok m r*
⟨*proof*⟩

lemma *to_mregex_ok*: *to_mregex r* = (*mr*, *ys*) ⟹ *set ys* = *atms r* ∧ *ok* (*length ys*) *mr*
⟨*proof*⟩

lemma *from_mregex_shift*: *from_mregex* (*shift r* (*length xs*)) (*xs* @ *ys*) = *from_mregex r ys*
⟨*proof*⟩

lemma *from_mregex_to_mregex*: *safe_regex m g r* ⟹ *case_prod from_mregex* (*to_mregex r*) = *r*
⟨*proof*⟩

lemma *from_mregex_eq*: *safe_regex m g r* ⟹ *to_mregex r* = (*mr*, *φs*) ⟹ *from_mregex mr φs* = *r*
⟨*proof*⟩

lemma *from_mregex_to_mregex_exec*: *safe_regex m g r* ⟹ *case_prod from_mregex* (*to_mregex_exec r xs*) = *r*
⟨*proof*⟩

derive *linorder mregex*

6.2.1 LPD

definition *saturate* **where**
saturate f = *while* (λ*S*. *f S* ≠ *S*) *f*

lemma *saturate_code*[*code*]:
saturate f S = (*let S′* = *f S in if S′* = *S then S else saturate f S′*)
⟨*proof*⟩

definition *MTimesL r S* = *MTimes r* ‘ *S*
definition *MTimesR R s* = (λ*r*. *MTimes r s*) ‘ *R*

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

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

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

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

lemma *LPDs_refl*: $r \in LPDs\ r$
  ⟨*proof*⟩
lemma *LPDs_trans*: $r \in LPD\ s \Longrightarrow s \in LPDs\ t \Longrightarrow r \in LPDs\ t$
  ⟨*proof*⟩

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

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

lemma *LPDi_MSkip*: $LPDi\ i\ (MSkip\ n) \subseteq MSkip\ `\ \{i.\ i \leq n\}$
  ⟨*proof*⟩

lemma *LPDs_MSkip*: $LPDs\ (MSkip\ n) \subseteq MSkip\ `\ \{i.\ i \leq n\}$
  ⟨*proof*⟩

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

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

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*⟩

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

lemma *LPDi_Star*: $j \leq i \Longrightarrow LPDi\ j\ (MStar\ r) \subseteq \{MStar\ r\} \cup MTimesR\ (\bigcup j \leq i.\ LPDi\ j\ r)\ (MStar\ r)$
⟨*proof*⟩

lemma *LPDs_Star*: *LPDs* (*MStar r*) ⊆ {*MStar r*} ∪ *MTimesR* (*LPDs r*) (*MStar r*)
⟨*proof*⟩

lemma *finite_LPDs*: *finite* (*LPDs r*)
⟨*proof*⟩

context begin

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

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

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

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

end

6.2.2 RPD

primrec *RPD* **where**
RPD (*MSkip n*) = (*case n of 0* ⇒ {} | *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* = (⋃ *s* ∈ *RPDi i r*. *RPD s*)
⟨*proof*⟩

definition *RPDs r* = (⋃ *i*. *RPDi i r*)

lemma *RPDs_refl*: *r* ∈ *RPDs r*
⟨*proof*⟩
lemma *RPDs_trans*: *r* ∈ *RPD s* ⟹ *s* ∈ *RPDs t* ⟹ *r* ∈ *RPDs t*
⟨*proof*⟩

lemma *RPDi_Test*:
RPDi i (*MSkip 0*) ⊆ {*MSkip 0*}
RPDi i (*MTestPos* φ) ⊆ {*MTestPos* φ}
RPDi i (*MTestNeg* φ) ⊆ {*MTestNeg* φ}
⟨*proof*⟩

lemma *RPDs_Test*:
RPDs (*MSkip 0*) ⊆ {*MSkip 0*}
RPDs (*MTestPos* φ) ⊆ {*MTestPos* φ}
RPDs (*MTestNeg* φ) ⊆ {*MTestNeg* φ}
⟨*proof*⟩

lemma *RPDi_MSkip*: *RPDi i* (*MSkip n*) ⊆ *MSkip* ‘ {*i*. *i* ≤ *n*}
⟨*proof*⟩

lemma *RPDs_MSkip*: *RPDs* (*MSkip n*) ⊆ *MSkip* ‘ {*i*. *i* ≤ *n*}
⟨*proof*⟩

lemma *RPDi_Plus*: *RPDi i* (*MPlus r s*) ⊆ {*MPlus r s*} ∪ *RPDi i r* ∪ *RPDi i s*
⟨*proof*⟩

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

lemma *RPDs_Plus*: *RPDs* (*MPlus r s*) ⊆ {*MPlus r s*} ∪ *RPDs r* ∪ *RPDs s*
⟨*proof*⟩

lemma *RPDi_Times*:
RPDi i (*MTimes r s*) ⊆ {*MTimes r s*} ∪ *MTimesL r* (⋃*j* ≤ *i*. *RPDi j s*) ∪ (⋃*j* ≤ *i*. *RPDi j r*)
⟨*proof*⟩

lemma *RPDs_Times*: *RPDs* (*MTimes r s*) ⊆ {*MTimes r s*} ∪ *MTimesL r* (*RPDs s*) ∪ *RPDs r*
⟨*proof*⟩

lemma *RPDi_Star*: *j* ≤ *i* ⟹ *RPDi j* (*MStar r*) ⊆ {*MStar r*} ∪ *MTimesL* (*MStar r*) (⋃*j* ≤ *i*. *RPDi j r*)
⟨*proof*⟩

lemma *RPDs_Star*: *RPDs* (*MStar r*) ⊆ {*MStar r*} ∪ *MTimesL* (*MStar r*) (*RPDs r*)
⟨*proof*⟩

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

context begin

private abbreviation (*input*) *addRPD r* ≡ λ*S*. *insert r S* ∪ *Set.bind* (*insert r S*) *RPD*

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

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

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

end

6.3 The executable monitor

type_synonym *ts* = *nat*

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

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

datatype *mconstraint = MEq | MLess | MLessEq*

record *args =*
  args_ivl :: 𝓘
  args_n :: nat
  args_L :: nat set
  args_R :: nat set
  args_pos :: bool

datatype *(dead 'msaux, dead 'muaux) mformula =*
  MRel event_data table
  | MPred Formula.name Formula.trm list
  | MLet Formula.name nat ('msaux, 'muaux) mformula ('msaux, 'muaux) mformula
  | MAnd nat set ('msaux, 'muaux) mformula bool nat set ('msaux, 'muaux) mformula event_data mbuf2
  | MAndAssign ('msaux, 'muaux) mformula nat × Formula.trm
  | MAndRel ('msaux, 'muaux) mformula Formula.trm × bool × mconstraint × Formula.trm
  | MAnds nat set list nat set list ('msaux, 'muaux) mformula list event_data mbufn
  | MOr ('msaux, 'muaux) mformula ('msaux, 'muaux) mformula event_data mbuf2
  | MNeg ('msaux, 'muaux) mformula
  | MExists ('msaux, 'muaux) mformula
  | MAgg bool nat Formula.agg_op nat Formula.trm ('msaux, 'muaux) mformula
  | MPrev 𝓘 ('msaux, 'muaux) mformula bool event_data table list ts list
  | MNext 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 𝓘 mregex mregex list ('msaux, 'muaux) mformula list event_data mbufn ts list event_data mrδaux
  | MMatchF 𝓘 mregex mregex list ('msaux, 'muaux) mformula list event_data mbufn ts list event_data mlδaux

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

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

lemma *regex_atms_size: x ∈ regex.atms r ⟹ size x < regex.size_regex size r*
  ⟨*proof*⟩

lemma *atms_size:*
  assumes *x ∈ atms r*
  shows *size x < Regex.size_regex size r*
⟨*proof*⟩

definition *init_args :: 𝓘 ⇒ nat ⇒ nat set ⇒ nat set ⇒ bool ⇒ args* **where**
  init_args I n L R pos = (|args_ivl = I, args_n = n, args_L = L, args_R = R, args_pos = pos|)

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

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

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

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

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

fun *eval_until* :: *I* ⇒ *ts* ⇒ *event_data muaux* ⇒ *event_data table list* × *event_data muaux* **where**
  eval_until I nt [] = ([], [])
| *eval_until I nt* ((*t*, *a1*, *a2*) # *aux*) = (*if t* + *right I* < *nt then*
    (*let* (*xs*, *aux*) = *eval_until I nt aux in* (*a2* # *xs*, *aux*)) *else* ([], (*t*, *a1*, *a2*) # *aux*))

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

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

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

⟨*proof*⟩

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

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

fun *split_assignment* :: *nat set* ⇒ *Formula.formula* ⇒ *nat* × *Formula.trm* **where**
  split_assignment X (*Formula.Eq t1 t2*) = (*case* (*t1*, *t2*) *of*
      (*Formula.Var x*, *Formula.Var y*) ⇒ *if x* ∈ *X then* (*y*, *t1*) *else* (*x*, *t2*)
    | (*Formula.Var x*, _) ⇒ (*x*, *t2*)
    | (_, *Formula.Var y*) ⇒ (*y*, *t1*))
| *split_assignment* _ _ = *undefined*

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

function (**in** *maux*) (*sequential*) *minit0* :: *nat* ⇒ *Formula.formula* ⇒ (*'msaux*, *'muaux*) *mformula* **where**
  minit0 n (*Formula.Neg* φ) = (*if fv* φ = {} *then MNeg* (*minit0 n* φ) *else MRel empty_table*)
| *minit0 n* (*Formula.Eq t1 t2*) = *MRel* (*eq_rel n t1 t2*)
| *minit0 n* (*Formula.Pred e ts*) = *MPred e ts*
| *minit0 n* (*Formula.Let p* φ ψ) = *MLet p* (*Formula.nfv* φ) (*minit0* (*Formula.nfv* φ) φ) (*minit0 n* ψ)

```
| minit0 n (Formula.Or 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
  ⟨proof⟩
termination (in maux)
  ⟨proof⟩

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

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

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

fun *mbuf2_add* :: *event_data table list* ⇒ *event_data table list* ⇒ *event_data mbuf2* ⇒ *event_data mbuf2*
where
mbuf2_add xs′ ys′ (xs, ys) = (xs @ xs′, ys @ ys′)

fun *mbuf2_take* :: (*event_data table* ⇒ *event_data table* ⇒ *′b*) ⇒ *event_data mbuf2* ⇒ *′b list* × *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* ⇒ *event_data table* ⇒ *ts* ⇒ *′b* ⇒ *′b*) ⇒ *′b* ⇒
event_data mbuf2 ⇒ *ts list* ⇒ *′b* × *event_data mbuf2* × *ts list* **where**
mbuf2t_take f z (x # xs, y # ys) (t # ts) = mbuf2t_take f (f x y t z) (xs, ys) ts
| *mbuf2t_take f z (xs, ys) ts = (z, (xs, ys), ts)*

lemma *size_list_length_diff1*: *xs ≠ [] ⟹ [] ∉ set xs ⟹*
size_list (λxs. length xs − Suc 0) xs < size_list length xs
⟨*proof*⟩

fun *mbufn_add* :: *event_data table list list* ⇒ *event_data mbufn* ⇒ *event_data mbufn* **where**
mbufn_add xs′ xs = List.map2 (@) xs xs′

function *mbufn_take* :: (*event_data table list* ⇒ *′b* ⇒ *′b*) ⇒ *′b* ⇒ *event_data mbufn* ⇒ *′b* × *event_data mbufn* **where**
mbufn_take f z buf = (if buf = [] ∨ [] ∈ set buf then (z, buf)
else mbufn_take f (f (map hd buf) z) (map tl buf))
⟨*proof*⟩
termination ⟨*proof*⟩

fun *mbufnt_take* :: (*event_data table list* ⇒ *ts* ⇒ *′b* ⇒ *′b*) ⇒
′b ⇒ *event_data mbufn* ⇒ *ts list* ⇒ *′b* × *event_data mbufn* × *ts list* **where**
mbufnt_take f z buf ts =
(if [] ∈ set buf ∨ 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* ⇒ *event_data list* ⇒ (*nat* ⇀ *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 maux) update_since :: args ⇒ event_data table ⇒ event_data table ⇒ ts ⇒
  'msaux ⇒ event_data table × 'msaux where
  update_since args rel1 rel2 nt aux =
    (let aux0 = join_msaux args rel1 (add_new_ts_msaux args nt aux);
         aux' = add_new_table_msaux args rel2 aux0
    in (result_msaux args aux', aux'))

definition lookup = Mapping.lookup_default empty_table

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

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

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

fun rδ :: (mregex ⇒ mregex) ⇒ (mregex, 'a table) mapping ⇒ 'a table list ⇒ mregex ⇒ 'a table where
  rδ κ X φs (MSkip n) = (case n of 0 ⇒ empty_table | Suc m ⇒ lookup X (κ (MSkip m)))
| rδ κ X φs (MTestPos i) = empty_table
| rδ κ X φs (MTestNeg i) = empty_table
| rδ κ X φs (MPlus r s) = rδ κ X φs r ∪ rδ κ X φs s
| rδ κ X φs (MTimes r s) = rδ (λt. κ (MTimes r t)) X φs s ∪ ε_lax (rδ κ X φs r) φs s
| rδ κ X φs (MStar r) = rδ (λt. κ (MTimes (MStar r) t)) X φs r

fun lδ :: (mregex ⇒ mregex) ⇒ (mregex, 'a table) mapping ⇒ 'a table list ⇒ mregex ⇒ 'a table where
  lδ κ X φs (MSkip n) = (case n of 0 ⇒ empty_table | Suc m ⇒ lookup X (κ (MSkip m)))
| lδ κ X φs (MTestPos i) = empty_table
| lδ κ X φs (MTestNeg i) = empty_table
| lδ κ X φs (MPlus r s) = lδ κ X φs r ∪ lδ κ X φs s
| lδ κ X φs (MTimes r s) = lδ (λt. κ (MTimes t s)) X φs r ∪ ε_lax (lδ κ X φs s) φs r
| lδ κ X φs (MStar r) = lδ (λt. κ (MTimes t (MStar r))) X φs r

lift_definition mrtabulate :: mregex list ⇒ (mregex ⇒ 'b table) ⇒ (mregex, 'b table) mapping
  is λks f. (map_of (List.map_filter (λk. let fk = f k in if fk = empty_table then None else Some (k,
```

fk)) *ks*)) ⟨*proof*⟩

lemma *lookup_tabulate*:
distinct xs ⟹ *lookup* (*mrtabulate xs f*) *x* = (*if x* ∈ *set xs then f x else empty_table*)
⟨*proof*⟩

definition *update_matchP* :: *nat* ⇒ *I* ⇒ *mregex* ⇒ *mregex list* ⇒ *event_data table list* ⇒ *ts* ⇒
event_data mrδaux ⇒ *event_data table* × *event_data mrδaux* **where**
update_matchP n I mr mrs rels nt aux =
(*let aux* = (*case* [(*t*, *mrtabulate mrs* (λ*mr*.
rδ id rel rels mr ∪ (*if t* = *nt then rε_strict n rels mr else* {}))).
(*t*, *rel*) ← *aux*, *enat* (*nt* − *t*) ≤ *right I*]
of [] ⇒ [(*nt*, *mrtabulate mrs* (*rε_strict n rels*))]
| *x* # *aux*′ ⇒ (*if fst x* = *nt then x* # *aux*′
else (*nt*, *mrtabulate mrs* (*rε_strict n rels*)) # *x* # *aux*′))
in (*foldr* (∪) [*lookup rel mr*. (*t*, *rel*) ← *aux*, *left I* ≤ *nt* − *t*] {}, *aux*))

definition *update_matchF_base* **where**
update_matchF_base n I mr mrs rels nt =
(*let X* = *mrtabulate mrs* (*lε_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 = (λ(*t*, *rels*′, *rel*) (*aux*′, *X*).
(*let Y* = *mrtabulate mrs* (*lδ id X rels*′)
in ((*t*, *rels*′, *if mem* (*nt* − *t*) *I then rel* ∪ *lookup Y mr else rel*) # *aux*′, *Y*)))

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

(*let* (*xss*, *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 (*List.enumerate* (*mstate_i st*) *xs*,
(|*mstate_i* = *mstate_i st* + *length xs*, *mstate_m* = *m*, *mstate_n* = *mstate_n st*|)))

6.4 Verdict delay

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

fun *progress* :: (*Formula.name* ⇀ *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 φ ψ*) *j* = *progress* (*P*(*p* ↦ *progress P φ j*)) *ψ 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 φ*) *j* = *progress P φ j*
| *progress P* (*Formula.Or φ ψ*) *j* = *min* (*progress P φ j*) (*progress P ψ j*)
| *progress P* (*Formula.And φ ψ*) *j* = *min* (*progress P φ j*) (*progress P ψ j*)
| *progress P* (*Formula.Ands l*) *j* = (*if l* = [] *then j else Min* (*set* (*map* (λ*φ*. *progress P φ j*) *l*)))
| *progress P* (*Formula.Exists φ*) *j* = *progress P φ j*
| *progress P* (*Formula.Agg y ω b f φ*) *j* = *progress P φ j*
| *progress P* (*Formula.Prev I φ*) *j* = (*if j* = *0 then 0 else min* (*Suc* (*progress P φ j*)) *j*)
| *progress P* (*Formula.Next I φ*) *j* = *progress P φ j* − *1*
| *progress P* (*Formula.Since φ I ψ*) *j* = *min* (*progress P φ j*) (*progress P ψ j*)
| *progress P* (*Formula.Until φ I ψ*) *j* =
Inf {*i*. ∀*k*. *k* < *j* ∧ *k* ≤ *min* (*progress P φ j*) (*progress P ψ 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* =
Inf {*i*. ∀*k*. *k* < *j* ∧ *k* ≤ *min_regex_default* (*progress P*) *r j* ⟶ *τ σ i* + *right I* ≥ *τ σ k*}

definition *progress_regex P* = *min_regex_default* (*progress P*)

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

end

definition *pred_mapping Q* = *pred_fun* (λ_. *True*) (*pred_option Q*)
definition *rel_mapping Q* = *rel_fun* (=) (*rel_option Q*)

lemma *pred_mapping_alt*: *pred_mapping Q P* = (∀ *p* ∈ *dom P*. *Q* (*the* (*P p*)))
⟨*proof*⟩

lemma *rel_mapping_alt*: *rel_mapping Q P P′* = (*dom P* = *dom P′* ∧ (∀ *p* ∈ *dom P*. *Q* (*the* (*P p*)) (*the* (*P′ p*))))
⟨*proof*⟩

lemma *rel_mapping_map_upd*[*simp*]: *Q x y* ⟹ *rel_mapping Q P P′* ⟹ *rel_mapping Q* (*P*(*p* ↦ *x*)) (*P′*(*p* ↦ *y*))
⟨*proof*⟩

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

lemma *pred_mapping_empty*[*simp*]: *pred_mapping* Q *Map.empty*
⟨*proof*⟩

lemma *pred_mapping_mono*: *pred_mapping* $Q\ P \Longrightarrow Q \leq R \Longrightarrow$ *pred_mapping* $R\ P$
⟨*proof*⟩

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

lemma *progress_mono_gen*: $j \leq j' \Longrightarrow$ *rel_mapping* $(\leq)\ P\ P' \Longrightarrow$ *progress* $\sigma\ P\ \varphi\ j \leq$ *progress* $\sigma\ P'\ \varphi\ j'$
⟨*proof*⟩

lemma *rel_mapping_reflp*: *reflp* $Q \Longrightarrow$ *rel_mapping* $Q\ P\ P$
⟨*proof*⟩

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

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

lemma *progress_le*: *progress* σ *Map.empty* $\varphi\ j \leq j$
⟨*proof*⟩

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

lemma *progress_0*[*simp*]:
progress σ *Map.empty* $\varphi\ 0 = 0$
⟨*proof*⟩

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*) ⇒ *None*
| (*Some* x, *None*) ⇒ *None*
| (*None*, *Some* x) ⇒ *None*
| (*Some* x, *Some* y) ⇒ *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* ∩ *dom P2*
⟨*proof*⟩

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

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)
⟨*proof*⟩

lemma *rel_mapping_trans*: $P\ OO\ Q \leq R \Longrightarrow$

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

abbreviation *range_mapping* :: *nat* ⇒ *nat* ⇒ (′*b* ⇒ *nat option*) ⇒ *bool* **where**
range_mapping i j P ≡ *pred_mapping* (λ*x*. *i* ≤ *x* ∧ *x* ≤ *j*) *P*

lemma *range_mapping_relax*:
range_mapping i j P $\Longrightarrow$ *i′* ≤ *i* $\Longrightarrow$ *j′* ≥ *j* $\Longrightarrow$ *range_mapping i′ j′ P*
⟨*proof*⟩

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

lemma *range_mapping_Max_mapping*[*simp*]:
finite X $\Longrightarrow$ *X* ≠ {} $\Longrightarrow$ ∀ *x*∈*X*. *range_mapping i* (*j x*) (*P x*) $\Longrightarrow$ *range_mapping i* (*Max* (*j* ‘ *X*)) (*Max_mapping* (*P* ‘ *X*))
⟨*proof*⟩

lemma *pred_mapping_le*:
pred_mapping ((≤) *i*) *P1* $\Longrightarrow$ *rel_mapping* (≤) *P1 P2* $\Longrightarrow$ *pred_mapping* ((≤) (*i* :: *nat*)) *P2*
⟨*proof*⟩

lemma *pred_mapping_le′*:
pred_mapping ((≤) *j*) *P1* $\Longrightarrow$ *i* ≤ *j* $\Longrightarrow$ *rel_mapping* (≤) *P1 P2* $\Longrightarrow$ *pred_mapping* ((≤) (*i* :: *nat*)) *P2*
⟨*proof*⟩

lemma *max_mapping_cobounded1*: *dom P1* ⊆ *dom P2* $\Longrightarrow$ *rel_mapping* (≤) *P1* (*max_mapping P1 P2*)
⟨*proof*⟩

lemma *max_mapping_cobounded2*: *dom P2* ⊆ *dom P1* $\Longrightarrow$ *rel_mapping* (≤) *P2* (*max_mapping P1 P2*)
⟨*proof*⟩

lemma *max_mapping_fun_upd2*[*simp*]:
max_mapping P1 (*P2*(*p* := *y*))(*p* ↦ *x*) = (*max_mapping P1 P2*)(*p* ↦ *x*)
⟨*proof*⟩

lemma *rel_mapping_max_mapping_fun_upd*: *dom P2* ⊆ *dom P1* $\Longrightarrow$ *p* ∈ *dom P2* $\Longrightarrow$ *the* (*P2 p*) ≤ *y* $\Longrightarrow$
rel_mapping (≤) *P2* (*max_mapping P1 P2*(*p* ↦ *y*))
⟨*proof*⟩

lemma *progress_ge_gen*: *Formula.future_bounded* φ $\Longrightarrow$
∃ *P j*. *dom P* = *S* ∧ *range_mapping i j P* ∧ *i* ≤ *progress* σ *P* φ *j*
⟨*proof*⟩

lemma *progress_ge*: *Formula.future_bounded* φ $\Longrightarrow$ ∃ *j*. *i* ≤ *progress* σ *Map.empty* φ *j*
⟨*proof*⟩

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

lemma *progress_time_conv*:
assumes ∀ *i*<*j*. τ σ *i* = τ σ′ *i*

shows *progress* σ P φ j = *progress* σ' P φ j
⟨*proof*⟩

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

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

lemma *bounded_rtranclp_mono*:
fixes n :: $'x$ :: *linorder*
assumes $\bigwedge i\ j.\ R\ i\ j \Longrightarrow j < n \Longrightarrow S\ i\ j \bigwedge i\ j.\ R\ i\ j \Longrightarrow i \leq j$
shows *rtranclp* R i $j \Longrightarrow j < n \Longrightarrow$ *rtranclp* S i j
⟨*proof*⟩

lemma *sat_prefix_conv_gen*:
assumes *prefix_of* π σ **and** *prefix_of* π σ'
shows $i <$ *progress* σ P φ (*plen* π) $\Longrightarrow$ *dom* V = *dom* $V' \Longrightarrow$ *dom* P = *dom* $V \Longrightarrow$
pred_mapping ($\lambda x.\ x \leq$ *plen* π) $P \Longrightarrow$
($\bigwedge p\ i\ \varphi.\ p \in$ *dom* $V \Longrightarrow i <$ *the* (P p) $\Longrightarrow$ *the* (V p) i = *the* (V' p) i) $\Longrightarrow$
Formula.sat σ V v i $\varphi \longleftrightarrow$ *Formula.sat* σ' V' v i φ
⟨*proof*⟩

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

lemma *progress_remove_neg*[*simp*]: *progress* σ P (*remove_neg* φ) j = *progress* σ P φ j
⟨*proof*⟩

lemma *safe_progress_get_and*: *safe_formula* $\varphi \Longrightarrow$
Min ((λφ. *progress* σ P φ j) ' *set* (*get_and_list* φ)) = *progress* σ P φ j
⟨*proof*⟩

lemma *progress_convert_multiway*: *safe_formula* $\varphi \Longrightarrow$ *progress* σ P (*convert_multiway* φ) j = *progress* σ P φ j
⟨*proof*⟩

6.5 Specification

definition *pprogress* :: *Formula.formula* ⇒ *Formula.prefix* ⇒ *nat* **where**
pprogress φ π = (*THE n.* $\forall \sigma$. *prefix_of* π $\sigma \longrightarrow$ *progress* σ *Map.empty* φ (*plen* π) = n)

lemma *pprogress_eq*: *prefix_of* π $\sigma \Longrightarrow$ *pprogress* φ π = *progress* σ *Map.empty* φ (*plen* π)
⟨*proof*⟩

locale *future_bounded_mfodl* =
fixes φ :: *Formula.formula*
assumes *future_bounded*: *Formula.future_bounded* φ

sublocale *future_bounded_mfodl* ⊆ *sliceable_timed_progress Formula.nfv* φ *Formula.fv* φ *relevant_events* φ
λσ v i. *Formula.sat* σ *Map.empty* v i φ *pprogress* φ
⟨*proof*⟩

locale *verimon_spec* =
  fixes φ :: *Formula.formula*
  assumes *monitorable*: *mmonitorable* φ

sublocale *verimon_spec* $\subseteq$ *future_bounded_mfodl*
  ⟨*proof*⟩

6.6 Correctness

6.6.1 Invariants

definition *wf_mbuf2* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* ⇒ *event_data table* ⇒ *bool*) ⇒ (*nat* ⇒ *event_data table* ⇒ *bool*) ⇒
    event_data mbuf2 ⇒ *bool* **where**
  wf_mbuf2 i ja jb P Q buf ⟷ *i* ≤ *ja* ∧ *i* ≤ *jb* ∧ (*case buf of* (*xs*, *ys*) ⇒
    list_all2 P [*i*..<*ja*] *xs* ∧ *list_all2 Q* [*i*..<*jb*] *ys*)

inductive *list_all3* :: (′*a* ⇒ ′*b* ⇒ ′*c* ⇒ *bool*) ⇒ ′*a list* ⇒ ′*b list* ⇒ ′*c list* ⇒ *bool* **for** *P* :: (′*a* ⇒ ′*b* ⇒ ′*c* ⇒ *bool*) **where**
  list_all3 P [] [] []
| *P a1 a2 a3* ⟹ *list_all3 P q1 q2 q3* ⟹ *list_all3 P* (*a1* # *q1*) (*a2* # *q2*) (*a3* # *q3*)

lemma *list_all3_list_all2D*: *list_all3 P xs ys zs* ⟹
  (*length xs* = *length ys* ∧ *list_all2* (*case_prod P*) (*zip xs ys*) *zs*)
  ⟨*proof*⟩

lemma *list_all2_list_all3I*: *length xs* = *length ys* ⟹ *list_all2* (*case_prod P*) (*zip xs ys*) *zs* ⟹
  list_all3 P xs ys zs
  ⟨*proof*⟩

lemma *list_all3_list_all2_eq*: *list_all3 P xs ys zs* ⟷
  (*length xs* = *length ys* ∧ *list_all2* (*case_prod P*) (*zip xs ys*) *zs*)
  ⟨*proof*⟩

lemma *list_all3_mapD*: *list_all3 P* (*map f xs*) (*map g ys*) (*map h zs*) ⟹
  list_all3 (λ*x y z*. *P* (*f x*) (*g y*) (*h z*)) *xs ys zs*
  ⟨*proof*⟩

lemma *list_all3_mapI*: *list_all3* (λ*x y z*. *P* (*f x*) (*g y*) (*h z*)) *xs ys zs* ⟹
  list_all3 P (*map f xs*) (*map g ys*) (*map h zs*)
  ⟨*proof*⟩

lemma *list_all3_map_iff*: *list_all3 P* (*map f xs*) (*map g ys*) (*map h zs*) ⟷
  list_all3 (λ*x y z*. *P* (*f x*) (*g y*) (*h z*)) *xs ys zs*
  ⟨*proof*⟩

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* ∧ *length ys* = *length zs* ∧ (∀ *i* < *length xs*. *P* (*xs*!*i*) (*ys*!*i*) (*zs*!*i*)))
  ⟨*proof*⟩

lemma *list_all3_refl* [*intro?*]:
  (⋀*x*. *x* ∈ *set xs* ⟹ *P x x x*) ⟹ *list_all3 P xs xs xs*

⟨*proof*⟩

definition *wf_mbufn* :: *nat* ⇒ *nat list* ⇒ (*nat* ⇒ *event_data table* ⇒ *bool*) *list* ⇒ *event_data mbufn* ⇒ *bool* **where**
wf_mbufn i js Ps buf ⟷ *list_all3* (λ*P j xs*. *i* ≤ *j* ∧ *list_all2 P* [*i*..<*j*] *xs*) *Ps js buf*

definition *wf_mbuf2'* :: *Formula.trace* ⇒ _ ⇒ _ ⇒ *nat* ⇒ *nat* ⇒ *event_data list set* ⇒
Formula.formula ⇒ *Formula.formula* ⇒ *event_data mbuf2* ⇒ *bool* **where**
wf_mbuf2' σ *P V j n R* φ ψ *buf* ⟷ *wf_mbuf2* (*min* (*progress* σ *P* φ *j*) (*progress* σ *P* ψ *j*))
(*progress* σ *P* φ *j*) (*progress* σ *P* ψ *j*)
(λ*i*. *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* φ))
(λ*i*. *qtable n* (*Formula.fv* ψ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* ψ)) *buf*

definition *wf_mbufn'* :: *Formula.trace* ⇒ _ ⇒ _ ⇒ *nat* ⇒ *nat* ⇒ *event_data list set* ⇒
Formula.formula Regex.regex ⇒ *event_data mbufn* ⇒ *bool* **where**
wf_mbufn' σ *P V j n R r buf* ⟷ (*case to_mregex r of* (*mr*, φ*s*) ⇒
wf_mbufn (*progress_regex* σ *P r j*) (*map* (λφ. *progress* σ *P* φ *j*) φ*s*)
(*map* (λφ *i*. *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* φ)) φ*s*)
buf)

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

definition *wf_ts* :: *Formula.trace* ⇒ _ ⇒ *nat* ⇒ *Formula.formula* ⇒ *Formula.formula* ⇒ *ts list* ⇒ *bool* **where**
wf_ts σ *P j* φ ψ *ts* ⟷ *list_all2* (λ*i t*. *t* = τ σ *i*) [*min* (*progress* σ *P* φ *j*) (*progress* σ *P* ψ *j*)..<*j*] *ts*

definition *wf_ts_regex* :: *Formula.trace* ⇒ _ ⇒ *nat* ⇒ *Formula.formula Regex.regex* ⇒ *ts list* ⇒ *bool* **where**
wf_ts_regex σ *P j r ts* ⟷ *list_all2* (λ*i t*. *t* = τ σ *i*) [*progress_regex* σ *P r j*..<*j*] *ts*

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

definition (**in** *msaux*) *wf_since_aux* :: *Formula.trace* ⇒ _ ⇒ *event_data list set* ⇒ *args* ⇒
Formula.formula ⇒ *Formula.formula* ⇒ *'msaux* ⇒ *nat* ⇒ *bool* **where**
wf_since_aux σ *V R args* φ ψ *aux ne* ⟷ *Formula.fv* φ ⊆ *Formula.fv* ψ ∧ (∃ *cur auxlist*. *valid_msaux args cur aux auxlist* ∧
cur = (*if ne* = *0 then 0 else* τ σ (*ne* − *1*)) ∧
sorted_wrt (λ*x y*. *fst x* > *fst y*) *auxlist* ∧
(∀ *t X*. (*t*, *X*) ∈ *set auxlist* ⟶ *ne* ≠ *0* ∧ *t* ≤ τ σ (*ne* − *1*) ∧ τ σ (*ne* − *1*) − *t* ≤ *right* (*args_ivl args*) ∧ (∃ *i*. τ σ *i* = *t*) ∧
qtable (*args_n args*) (*Formula.fv* ψ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) (*ne*−*1*) (*Sincep* (*args_pos args*) φ (*point* (τ σ (*ne* − *1*) − *t*)) ψ)) *X*) ∧
(∀ *t*. *ne* ≠ *0* ∧ *t* ≤ τ σ (*ne* − *1*) ∧ τ σ (*ne* − *1*) − *t* ≤ *right* (*args_ivl args*) ∧ (∃ *i*. τ σ *i* = *t*) ⟶
(∃ *X*. (*t*, *X*) ∈ *set auxlist*)))

definition *wf_matchP_aux* :: *Formula.trace* ⇒ _ ⇒ *nat* ⇒ *event_data list set* ⇒
I ⇒ *Formula.formula Regex.regex* ⇒ *event_data mrδaux* ⇒ *nat* ⇒ *bool* **where**
wf_matchP_aux σ *V n R I r aux ne* ⟷ *sorted_wrt* (λ*x y*. *fst x* > *fst y*) *aux* ∧
(∀ *t X*. (*t*, *X*) ∈ *set aux* ⟶ *ne* ≠ *0* ∧ *t* ≤ τ σ (*ne*−*1*) ∧ τ σ (*ne*−*1*) − *t* ≤ *right I* ∧ (∃ *i*. τ σ *i* = *t*) ∧
(*case to_mregex r of* (*mr*, φ*s*) ⇒
(∀ *ms* ∈ *RPDs mr*. *qtable n* (*Formula.fv_regex r*) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) (*ne*−*1*)
(*Formula.MatchP* (*point* (τ σ (*ne*−*1*) − *t*)) (*from_mregex ms* φ*s*)))

(*lookup X ms*)))) ∧
(∀ *t*. *ne* ≠ *0* ∧ *t* ≤ *τ σ* (*ne*−*1*) ∧ *τ σ* (*ne*−*1*) − *t* ≤ *right I* ∧ (∃ *i*. *τ σ i* = *t*) ⟶
(∃ *X*. (*t*, *X*) ∈ *set aux*))

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

lemma (**in** *msaux*) *wf_since_aux_UNIV_alt*:
wf_since_aux σ V UNIV args φ ψ aux ne ⟷ *Formula.fv φ* ⊆ *Formula.fv ψ* ∧ (∃ *cur auxlist*. *valid_msaux args cur aux auxlist* ∧
cur = (*if ne* = *0 then 0 else τ σ* (*ne* − *1*)) ∧
sorted_wrt (λ*x y*. *fst x* > *fst y*) *auxlist* ∧
(∀ *t X*. (*t*, *X*) ∈ *set auxlist* ⟶ *ne* ≠ *0* ∧ *t* ≤ *τ σ* (*ne* − *1*) ∧ *τ σ* (*ne* − *1*) − *t* ≤ *right* (*args_ivl args*) ∧ (∃ *i*. *τ σ i* = *t*) ∧
wf_table (*args_n args*) (*Formula.fv ψ*)
(λ*v*. *Formula.sat σ V* (*map the v*) (*ne* − *1*) (*Sincep* (*args_pos args*) *φ* (*point* (*τ σ* (*ne* − *1*) − *t*)) *ψ*)) *X*) ∧
(∀ *t*. *ne* ≠ *0* ∧ *t* ≤ *τ σ* (*ne* − *1*) ∧ *τ σ* (*ne* − *1*) − *t* ≤ *right* (*args_ivl args*) ∧ (∃ *i*. *τ σ i* = *t*) ⟶
(∃ *X*. (*t*, *X*) ∈ *set auxlist*)))
⟨*proof*⟩

definition *wf_until_auxlist* :: *Formula.trace* ⇒ _ ⇒ *nat* ⇒ *event_data list set* ⇒ *bool* ⇒
Formula.formula ⇒ ℐ ⇒ *Formula.formula* ⇒ *event_data muaux* ⇒ *nat* ⇒ *bool* **where**
wf_until_auxlist σ V n R pos φ I ψ auxlist ne ⟷
list_all2 (λ*x i*. *case x of* (*t*, *r1*, *r2*) ⇒ *t* = *τ σ i* ∧
qtable n (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *if pos then* (∀ *k*∈{*i*..<*ne*+*length auxlist*}. *Formula.sat σ V* (*map the v*) *k φ*)
else (∃ *k*∈{*i*..<*ne*+*length auxlist*}. *Formula.sat σ V* (*map the v*) *k φ*)) *r1* ∧
qtable n (*Formula.fv ψ*) (*mem_restr R*) (λ*v*. (∃ *j*. *i* ≤ *j* ∧ *j* < *ne* + *length auxlist* ∧ *mem* (*τ σ j* − *τ σ i*) *I* ∧
Formula.sat σ V (*map the v*) *j ψ* ∧
(∀ *k*∈{*i*..<*j*}. *if pos then Formula.sat σ V* (*map the v*) *k φ else* ¬ *Formula.sat σ V* (*map the v*) *k φ*))) *r2*)
auxlist [*ne*..<*ne*+*length auxlist*]

definition (**in** *muaux*) *wf_until_aux* :: *Formula.trace* ⇒ _ ⇒ *event_data list set* ⇒ *args* ⇒
Formula.formula ⇒ *Formula.formula* ⇒ '*muaux* ⇒ *nat* ⇒ *bool* **where**
wf_until_aux σ V R args φ ψ aux ne ⟷ *Formula.fv φ* ⊆ *Formula.fv ψ* ∧
(∃ *cur auxlist*. *valid_muaux args cur aux auxlist* ∧
cur = (*if ne* + *length auxlist* = *0 then 0 else τ σ* (*ne* + *length auxlist* − *1*)) ∧
wf_until_auxlist σ V (*args_n args*) *R* (*args_pos args*) *φ* (*args_ivl args*) *ψ auxlist ne*)

lemma (**in** *muaux*) *wf_until_aux_UNIV_alt*:
wf_until_aux σ V UNIV args φ ψ aux ne ⟷ *Formula.fv φ* ⊆ *Formula.fv ψ* ∧
(∃ *cur auxlist*. *valid_muaux args cur aux auxlist* ∧
cur = (*if ne* + *length auxlist* = *0 then 0 else τ σ* (*ne* + *length auxlist* − *1*)) ∧
list_all2 (λ*x i*. *case x of* (*t*, *r1*, *r2*) ⇒ *t* = *τ σ i* ∧
wf_table (*args_n args*) (*Formula.fv φ*) (λ*v*. *if* (*args_pos args*)
then (∀ *k*∈{*i*..<*ne*+*length auxlist*}. *Formula.sat σ V* (*map the v*) *k φ*)
else (∃ *k*∈{*i*..<*ne*+*length auxlist*}. *Formula.sat σ V* (*map the v*) *k φ*)) *r1* ∧
wf_table (*args_n args*) (*Formula.fv ψ*) (λ*v*. ∃ *j*. *i* ≤ *j* ∧ *j* < *ne* + *length auxlist* ∧ *mem* (*τ σ j* − *τ σ i*) (*args_ivl args*) ∧
Formula.sat σ V (*map the v*) *j ψ* ∧
(∀ *k*∈{*i*..<*j*}. *if* (*args_pos args*) *then Formula.sat σ V* (*map the v*) *k φ else* ¬ *Formula.sat σ V* (*map the v*) *k φ*)) *r2*)
auxlist [*ne*..<*ne*+*length auxlist*])
⟨*proof*⟩

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

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

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

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

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

$\psi' = formula\_of\_constraint\ conf \Longrightarrow$
$(let\ (t1, \_, \_, t2) = conf\ in\ Formula.fv\_trm\ t1 \cup Formula.fv\_trm\ t2 \subseteq Formula.fv\ \varphi') \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MAndRel\ \varphi\ conf)\ (Formula.And\ \varphi'\ \psi')$
| *Ands*: $list\_all2\ (\lambda\varphi\ \varphi'.\ wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi')\ l\ (l\_pos\ @\ map\ remove\_neg\ l\_neg) \Longrightarrow$
$wf\_mbufn\ (progress\ \sigma\ P\ (Formula.Ands\ l')\ j)\ (map\ (\lambda\psi.\ progress\ \sigma\ P\ \psi\ j)\ (l\_pos\ @\ map\ remove\_neg\ l\_neg))\ (map\ (\lambda\psi\ i.$
$qtable\ n\ (Formula.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \psi))\ (l\_pos\ @\ map\ remove\_neg\ l\_neg))\ buf \Longrightarrow$
$(l\_pos, l\_neg) = partition\ safe\_formula\ l' \Longrightarrow$
$l\_pos \neq [] \Longrightarrow$
$list\_all\ safe\_formula\ (map\ remove\_neg\ l\_neg) \Longrightarrow$
$A\_pos = map\ fv\ l\_pos \Longrightarrow$
$A\_neg = map\ fv\ l\_neg \Longrightarrow$
$\bigcup(set\ A\_neg) \subseteq \bigcup(set\ A\_pos) \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MAnds\ A\_pos\ A\_neg\ l\ buf)\ (Formula.Ands\ l')$
| *Or*: $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi' \Longrightarrow wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \psi\ \psi' \Longrightarrow$
$Formula.fv\ \varphi' = Formula.fv\ \psi' \Longrightarrow$
$wf\_mbuf2'\ \sigma\ P\ V\ j\ n\ R\ \varphi'\ \psi'\ buf \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MOr\ \varphi\ \psi\ buf)\ (Formula.Or\ \varphi'\ \psi')$
| *Neg*: $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi' \Longrightarrow Formula.fv\ \varphi' = \{\} \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MNeg\ \varphi)\ (Formula.Neg\ \varphi')$
| *Exists*: $wf\_mformula\ \sigma\ j\ P\ V\ (Suc\ n)\ (lift\_envs\ R)\ \varphi\ \varphi' \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MExists\ \varphi)\ (Formula.Exists\ \varphi')$
| *Agg*: $wf\_mformula\ \sigma\ j\ P\ V\ (b + n)\ (lift\_envs'\ b\ R)\ \varphi\ \varphi' \Longrightarrow$
$y < n \Longrightarrow$
$y + b \notin Formula.fv\ \varphi' \Longrightarrow$
$\{0..<b\} \subseteq Formula.fv\ \varphi' \Longrightarrow$
$Formula.fv\_trm\ f \subseteq Formula.fv\ \varphi' \Longrightarrow$
$g0 = (Formula.fv\ \varphi' \subseteq \{0..<b\}) \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MAgg\ g0\ y\ \omega\ b\ f\ \varphi)\ (Formula.Agg\ y\ \omega\ b\ f\ \varphi')$
| *Prev*: $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi' \Longrightarrow$
$first \longleftrightarrow j = 0 \Longrightarrow$
$list\_all2\ (\lambda i.\ qtable\ n\ (Formula.fv\ \varphi')\ (mem\_restr\ R)\ (\lambda v.\ Formula.sat\ \sigma\ V\ (map\ the\ v)\ i\ \varphi'))$
$[min\ (progress\ \sigma\ P\ \varphi'\ j)\ (j-1)..<progress\ \sigma\ P\ \varphi'\ j]\ buf \Longrightarrow$
$list\_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [min\ (progress\ \sigma\ P\ \varphi'\ j)\ (j-1)..<j]\ nts \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MPrev\ I\ \varphi\ first\ buf\ nts)\ (Formula.Prev\ I\ \varphi')$
| *Next*: $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi' \Longrightarrow$
$first \longleftrightarrow progress\ \sigma\ P\ \varphi'\ j = 0 \Longrightarrow$
$list\_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [progress\ \sigma\ P\ \varphi'\ j - 1..<j]\ nts \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MNext\ I\ \varphi\ first\ nts)\ (Formula.Next\ I\ \varphi')$
| *Since*: $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi' \Longrightarrow wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \psi\ \psi' \Longrightarrow$
$if\ args\_pos\ args\ then\ \varphi'' = \varphi'\ else\ \varphi'' = Formula.Neg\ \varphi' \Longrightarrow$
$safe\_formula\ \varphi'' = args\_pos\ args \Longrightarrow$
$args\_ivl\ args = I \Longrightarrow$
$args\_n\ args = n \Longrightarrow$
$args\_L\ args = Formula.fv\ \varphi' \Longrightarrow$
$args\_R\ args = Formula.fv\ \psi' \Longrightarrow$
$Formula.fv\ \varphi' \subseteq Formula.fv\ \psi' \Longrightarrow$
$wf\_mbuf2'\ \sigma\ P\ V\ j\ n\ R\ \varphi'\ \psi'\ buf \Longrightarrow$
$wf\_ts\ \sigma\ P\ j\ \varphi'\ \psi'\ nts \Longrightarrow$
$wf\_since\_aux\ \sigma\ V\ R\ args\ \varphi'\ \psi'\ aux\ (progress\ \sigma\ P\ (Formula.Since\ \varphi''\ I\ \psi')\ j) \Longrightarrow$
$wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ (MSince\ args\ \varphi\ \psi\ buf\ nts\ aux)\ (Formula.Since\ \varphi''\ I\ \psi')$
| *Until*: $wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \varphi\ \varphi' \Longrightarrow wf\_mformula\ \sigma\ j\ P\ V\ n\ R\ \psi\ \psi' \Longrightarrow$
$if\ args\_pos\ args\ then\ \varphi'' = \varphi'\ else\ \varphi'' = Formula.Neg\ \varphi' \Longrightarrow$
$safe\_formula\ \varphi'' = args\_pos\ args \Longrightarrow$
$args\_ivl\ args = I \Longrightarrow$
$args\_n\ args = n \Longrightarrow$
$args\_L\ args = Formula.fv\ \varphi' \Longrightarrow$

args_R args = Formula.fv ψ' ⟹
Formula.fv φ' ⊆ Formula.fv ψ' ⟹
wf_mbuf2' σ P V j n R φ' ψ' buf ⟹
wf_ts σ P j φ' ψ' nts ⟹
wf_until_aux σ V R args φ' ψ' aux (progress σ P (Formula.Until φ'' I ψ') j) ⟹
progress σ P (Formula.Until φ'' I ψ') j + length_muaux args aux = min (progress σ P φ' j) (progress σ P ψ' j) ⟹
wf_mformula σ j P V n R (MUntil args φ ψ buf nts aux) (Formula.Until φ'' I ψ')
| *MatchP*: *(case to_mregex r of (mr', φs') ⇒*
list_all2 (wf_mformula σ j P V n R) φs φs' ∧ mr = mr') ⟹
mrs = sorted_list_of_set (RPDs mr) ⟹
safe_regex Past Strict r ⟹
wf_mbufn' σ P V j n R r buf ⟹
wf_ts_regex σ P j r nts ⟹
wf_matchP_aux σ V n R I r aux (progress σ P (Formula.MatchP I r) j) ⟹
wf_mformula σ j P V n R (MMatchP I mr mrs φs buf nts aux) (Formula.MatchP I r)
| *MatchF*: *(case to_mregex r of (mr', φs') ⇒*
list_all2 (wf_mformula σ j P V n R) φs φs' ∧ mr = mr') ⟹
mrs = sorted_list_of_set (LPDs mr) ⟹
safe_regex Futu Strict r ⟹
wf_mbufn' σ P V j n R r buf ⟹
wf_ts_regex σ P j r nts ⟹
wf_matchF_aux σ V n R I r aux (progress σ P (Formula.MatchF I r) j) 0 ⟹
progress σ P (Formula.MatchF I r) j + length aux = progress_regex σ P r j ⟹
wf_mformula σ j P V n R (MMatchF I mr mrs φs buf nts aux) (Formula.MatchF I r)

definition (**in** *maux*) *wf_mstate* :: *Formula.formula ⇒ Formula.prefix ⇒ event_data list set ⇒ ('msaux, 'muaux) mstate ⇒ bool* **where**
wf_mstate φ π R st ⟷ mstate_n st = Formula.nfv φ ∧ (∀σ. prefix_of π σ ⟶
mstate_i st = progress σ Map.empty φ (plen π) ∧
wf_mformula σ (plen π) Map.empty Map.empty (mstate_n st) R (mstate_m st) φ)

6.6.2 Initialisation

lemma *wf_mbuf2'_0*: *pred_mapping (λx. x = 0) P ⟹ wf_mbuf2' σ P V 0 n R φ ψ ([], [])*
⟨*proof*⟩

lemma *wf_mbufn'_0*: *to_mregex r = (mr, φs) ⟹ pred_mapping (λx. x = 0) P ⟹ wf_mbufn' σ P V 0 n R r (replicate (length φs) [])*
⟨*proof*⟩

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

lemma *wf_ts_regex_0*: *wf_ts_regex σ P 0 r []*
⟨*proof*⟩

lemma (**in** *msaux*) *wf_since_aux_Nil*: *Formula.fv φ' ⊆ Formula.fv ψ' ⟹*
wf_since_aux σ V R (init_args I n (Formula.fv φ') (Formula.fv ψ') b) φ' ψ' (init_msaux (init_args I n (Formula.fv φ') (Formula.fv ψ') b)) 0
⟨*proof*⟩

lemma (**in** *muaux*) *wf_until_aux_Nil*: *Formula.fv φ' ⊆ Formula.fv ψ' ⟹*
wf_until_aux σ V R (init_args I n (Formula.fv φ') (Formula.fv ψ') b) φ' ψ' (init_muaux (init_args I n (Formula.fv φ') (Formula.fv ψ') b)) 0
⟨*proof*⟩

lemma *wf_matchP_aux_Nil*: *wf_matchP_aux σ V n R I r [] 0*
⟨*proof*⟩

lemma *wf_matchF_aux_Nil*: *wf_matchF_aux* σ *V n R I r* [] *0 k*
⟨*proof*⟩

lemma *fv_regex_alt*: *safe_regex m g r* $\Longrightarrow$ *Formula.fv_regex r* = $(\bigcup \varphi \in$ *atms r*. *Formula.fv* $\varphi)$
⟨*proof*⟩

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

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

lemma (**in** *maux*) *wf_mstate_minit*: *safe_formula* $\varphi \Longrightarrow$ *wf_mstate* φ *pnil R* (*minit* φ)
⟨*proof*⟩

6.6.3 Evaluation

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

lemma *match_fvi_trm_None*: *Some f* = *match ts xs* $\Longrightarrow \forall t{\in}$*set ts*. $x \notin$ *Formula.fv_trm t* $\Longrightarrow$ *f x* = *None*
⟨*proof*⟩

lemma *match_fvi_trm_Some*: *Some f* = *match ts xs* $\Longrightarrow$ *t* $\in$ *set ts* $\Longrightarrow$ *x* $\in$ *Formula.fv_trm t* $\Longrightarrow$ *f x* $\neq$ *None*
⟨*proof*⟩

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

lemma *wf_tuple_tabulate_Some*: *wf_tuple n A* (*Table.tabulate f 0 n*) $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *x* $<$ *n* $\Longrightarrow \exists y.$ *f x* = *Some y*
⟨*proof*⟩

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

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

lemma *in_eq_rel*: *wf_tuple n* (*Formula.fv_trm t1* $\cup$ *Formula.fv_trm t2*) *v* $\Longrightarrow$
is_simple_eq t1 t2 $\Longrightarrow$
Formula.eval_trm (*map the v*) *t1* = *Formula.eval_trm* (*map the v*) *t2* $\Longrightarrow$
v $\in$ *eq_rel n t1 t2*
⟨*proof*⟩

lemma *table_eq_rel*: *is_simple_eq t1 t2* $\Longrightarrow$
table n (*Formula.fv_trm t1* $\cup$ *Formula.fv_trm t2*) (*eq_rel n t1 t2*)
⟨*proof*⟩

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

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

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

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

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

lemma *mem_restr_lift_envs'_append*[*simp*]:
length xs = *b* $\Longrightarrow$ *mem_restr* (*lift_envs' b R*) (*xs* @ *ys*) = *mem_restr R ys*
⟨*proof*⟩

lemma *nth_list_update_alt*: *xs*[*i* := *x*] ! *j* = (*if i* < *length xs* ∧ *i* = *j then x else xs* ! *j*)
⟨*proof*⟩

lemma *wf_tuple_upd_None*: *wf_tuple n A xs* $\Longrightarrow$ *A* − {*i*} = *B* $\Longrightarrow$ *wf_tuple n B* (*xs*[*i*:=*None*])
⟨*proof*⟩

lemma *mem_restr_upd_None*: *mem_restr R xs* $\Longrightarrow$ *mem_restr R* (*xs*[*i*:=*None*])
⟨*proof*⟩

lemma *mem_restr_dropI*: *mem_restr* (*lift_envs' b R*) *xs* $\Longrightarrow$ *mem_restr R* (*drop b xs*)
⟨*proof*⟩

lemma *mem_restr_dropD*:
assumes *b* ≤ *length xs* **and** *mem_restr R* (*drop b xs*)
shows *mem_restr* (*lift_envs' b R*) *xs*
⟨*proof*⟩

lemma *wf_tuple_append*: *wf_tuple a* {*x* ∈ *A*. *x* < *a*} *xs* $\Longrightarrow$
wf_tuple b {*x* − *a* | *x*. *x* ∈ *A* ∧ *x* ≥ *a*} *ys* $\Longrightarrow$
wf_tuple (*a* + *b*) *A* (*xs* @ *ys*)
⟨*proof*⟩

lemma *wf_tuple_map_Some*: *length xs* = *n* $\Longrightarrow$ {*0..<n*} ⊆ *A* $\Longrightarrow$ *wf_tuple n A* (*map Some xs*)
⟨*proof*⟩

lemma *wf_tuple_drop*: *wf_tuple* (*b* + *n*) *A xs* $\Longrightarrow$ {*x* − *b* | *x*. *x* ∈ *A* ∧ *x* ≥ *b*} = *B* $\Longrightarrow$
wf_tuple n B (*drop b xs*)
⟨*proof*⟩

lemma *ecard_image*: *inj_on f A* $\Longrightarrow$ *ecard* (*f* ‘ *A*) = *ecard A*
⟨*proof*⟩

lemma *meval_trm_eval_trm*: *wf_tuple n A x* $\Longrightarrow$ *fv_trm t* ⊆ *A* $\Longrightarrow$ ∀ *i*∈*A*. *i* < *n* $\Longrightarrow$

meval_trm t x = Formula.eval_trm (map the x) t
⟨*proof*⟩

lemma *list_update_id*: *xs ! i = z ⟹ xs[i:=z] = xs*
⟨*proof*⟩

lemma *qtable_wf_tupleD*: *qtable n A P Q X ⟹ ∀ x∈X. wf_tuple n A x*
⟨*proof*⟩

lemma *qtable_eval_agg*:
assumes *inner*: *qtable (b + n) (Formula.fv φ) (mem_restr (lift_envs' b R))*
(λv. Formula.sat σ V (map the v) i φ) rel
and *n*: *∀ x∈Formula.fv (Formula.Agg y ω b f φ). x < n*
and *fresh*: *y + b ∉ Formula.fv φ*
and *b_fv*: *{0..<b} ⊆ Formula.fv φ*
and *f_fv*: *Formula.fv_trm f ⊆ Formula.fv φ*
and *g0*: *g0 = (Formula.fv φ ⊆ {0..<b})*
shows *qtable n (Formula.fv (Formula.Agg y ω b f φ)) (mem_restr R)*
(λv. Formula.sat σ V (map the v) i (Formula.Agg y ω b f φ)) (eval_agg n g0 y ω b f rel)
(**is** *qtable _ ?fv _ ?Q ?rel'*)
⟨*proof*⟩

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*⟩

lemma *mnext*: *mprev_next I xs ts = (ys, xs', ts') ⟹*
list_all2 P [Suc i..<j'] xs ⟹ list_all2 (λi t. t = τ σ i) [i..<j] ts ⟹ Suc i ≤ j' ⟹ i < j ⟹
list_all2 (λi X. if mem (τ σ (Suc i) - τ σ i) I then P (Suc i) X else X = empty_table)
[i..<min (j'−1) (j−1)] ys ∧
list_all2 P [Suc (min (j'−1) (j−1))..<j'] xs' ∧
list_all2 (λi t. t = τ σ i) [min (j'−1) (j−1)..<j] ts'
⟨*proof*⟩

lemma *in_foldr_UnI*: *x ∈ A ⟹ A ∈ set xs ⟹ x ∈ foldr (∪) xs {}*
⟨*proof*⟩

lemma *in_foldr_UnE*: *x ∈ foldr (∪) xs {} ⟹ (⋀A. A ∈ set xs ⟹ x ∈ A ⟹ P) ⟹ P*
⟨*proof*⟩

lemma *sat_the_restrict*: *fv φ ⊆ A ⟹ Formula.sat σ V (map the (restrict A v)) i φ = Formula.sat σ V (map the v) i φ*
⟨*proof*⟩

lemma *eps_the_restrict*: *fv_regex r ⊆ A ⟹ Regex.eps (Formula.sat σ V (map the (restrict A v))) i r = Regex.eps (Formula.sat σ V (map the v)) i r*
⟨*proof*⟩

lemma *sorted_wrt_filter*[*simp*]: *sorted_wrt R xs ⟹ sorted_wrt R (filter P xs)*
⟨*proof*⟩

lemma *concat_map_filter*[*simp*]:
concat (map f (filter P xs)) = concat (map (λx. if P x then f x else []) xs)
⟨*proof*⟩

lemma *map_filter_alt*:
map f (*filter P xs*) = *concat* (*map* (λ*x*. *if P x then* [*f x*] *else* []) *xs*)
⟨*proof*⟩

lemma (**in** *maux*) *update_since*:
assumes *pre*: *wf_since_aux σ V R args φ ψ aux ne*
and *qtable1*: *qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *ne φ*) *rel1*
and *qtable2*: *qtable n* (*Formula.fv ψ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *ne ψ*) *rel2*
and *result_eq*: (*rel*, *aux'*) = *update_since args rel1 rel2* (*τ σ ne*) *aux*
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_since_aux σ V R args φ ψ aux'* (*Suc ne*)
and *qtable n* (*Formula.fv ψ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *ne* (*Sincep pos φ I ψ*)) *rel*
⟨*proof*⟩

lemma *fv_regex_from_mregex*:
ok (*length φs*) *mr* ⟹ *fv_regex* (*from_mregex mr φs*) ⊆ (⋃ *φ* ∈ *set φs*. *fv φ*)
⟨*proof*⟩

lemma *qtable_ε_lax*:
assumes *ok* (*length φs*) *mr*
and *list_all2* (λ*φ rel*. *qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ*) *rel*) *φs rels*
and *fv_regex* (*from_mregex mr φs*) ⊆ *A* **and** *qtable n A* (*mem_restr R*) *Q guard*
shows *qtable n A* (*mem_restr R*)
(λ*v*. *Regex.eps* (*Formula.sat σ V* (*map the v*)) *i* (*from_mregex mr φs*) ∧ *Q v*) (*ε_lax guard rels mr*)
⟨*proof*⟩

lemma *nullary_qtable_cases*: *qtable n* {} *P Q X* ⟹ (*X* = *empty_table* ∨ *X* = *unit_table n*)
⟨*proof*⟩

lemma *qtable_empty_unit_table*:
qtable n {} *R P empty_table* ⟹ *qtable n* {} *R* (λ*v*. ¬ *P v*) (*unit_table n*)
⟨*proof*⟩

lemma *qtable_unit_empty_table*:
qtable n {} *R P* (*unit_table n*) ⟹ *qtable n* {} *R* (λ*v*. ¬ *P v*) *empty_table*
⟨*proof*⟩

lemma *qtable_nonempty_empty_table*:
qtable n {} *R P X* ⟹ *x* ∈ *X* ⟹ *qtable n* {} *R* (λ*v*. ¬ *P v*) *empty_table*
⟨*proof*⟩

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

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

lemma *rtranclp_False*: (λ*i j. False*)** = (=)
⟨*proof*⟩

inductive *ok_rctxt* **for** *φs* **where**
ok_rctxt φs id id
| *ok_rctxt φs κ κ′* ⟹ *ok_rctxt φs* (λ*t. κ* (*MTimes mr t*)) (λ*t. κ′* (*Regex.Times* (*from_mregex mr φs*) *t*))

lemma *ok_rctxt_swap*: *ok_rctxt φs κ κ′* ⟹ *from_mregex* (*κ mr*) *φs* = *κ′* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *ok_rctxt_cong*: *ok_rctxt φs κ κ′* ⟹ *Regex.match* (*Formula.sat σ V v*) *r* = *Regex.match* (*Formula.sat σ V v*) *s* ⟹
Regex.match (*Formula.sat σ V v*) (*κ′ r*) *i j* = *Regex.match* (*Formula.sat σ V v*) (*κ′ s*) *i j*
⟨*proof*⟩

lemma *qtable_rδκ*:
assumes *ok* (*length φs*) *mr fv_regex* (*from_mregex mr φs*) ⊆ *A*
and *list_all2* (λ*φ rel. qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v. Formula.sat σ V* (*map the v*) *j φ*) *rel*) *φs rels*
and *ok_rctxt φs κ κ′*
and ∀ *ms* ∈ *κ* ‘ *RPD mr. qtable n A* (*mem_restr R*) (λ*v. Q* (*map the v*) (*from_mregex ms φs*)) (*lookup rel ms*)
shows *qtable n A* (*mem_restr R*)
(λ*v.* ∃ *s* ∈ *Regex.rpdκ κ′* (*Formula.sat σ V* (*map the v*)) *j* (*from_mregex mr φs*). *Q* (*map the v*) *s*)
(*rδ κ rel rels mr*)
⟨*proof*⟩

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

inductive *ok_lctxt* **for** *φs* **where**
ok_lctxt φs id id
| *ok_lctxt φs κ κ′* ⟹ *ok_lctxt φs* (λ*t. κ* (*MTimes t mr*)) (λ*t. κ′* (*Regex.Times t* (*from_mregex mr φs*)))

lemma *ok_lctxt_swap*: *ok_lctxt φs κ κ′* ⟹ *from_mregex* (*κ mr*) *φs* = *κ′* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *ok_lctxt_cong*: *ok_lctxt φs κ κ′* ⟹ *Regex.match* (*Formula.sat σ V v*) *r* = *Regex.match* (*Formula.sat σ V v*) *s* ⟹
Regex.match (*Formula.sat σ V v*) (*κ′ r*) *i j* = *Regex.match* (*Formula.sat σ V v*) (*κ′ s*) *i j*
⟨*proof*⟩

lemma *qtable_lδκ*:
assumes *ok* (*length φs*) *mr fv_regex* (*from_mregex mr φs*) ⊆ *A*
and *list_all2* (λ*φ rel. qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v. Formula.sat σ V* (*map the v*) *j φ*) *rel*) *φs rels*
and *ok_lctxt φs κ κ′*
and ∀ *ms* ∈ *κ* ‘ *LPD mr. qtable n A* (*mem_restr R*) (λ*v. Q* (*map the v*) (*from_mregex ms φs*)) (*lookup*

rel ms)
shows *qtable n A* (*mem_restr R*)
(λ*v*. ∃ *s* ∈ *Regex.lpdκ κ′* (*Formula.sat σ V* (*map the v*)) *j* (*from_mregex mr φs*). *Q* (*map the v*) *s*)
(*lδ κ rel rels mr*)
⟨*proof*⟩

lemmas *qtable_lδ* = *qtable_lδκ*[*OF _ _ _ ok_lctxt.intros*(*1*), *unfolded lpdκ_lpd image_id id_apply*]

lemma *RPD_fv_regex_le*:
ms ∈ *RPD mr* ⟹ *fv_regex* (*from_mregex ms φs*) ⊆ *fv_regex* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *RPD_safe*: *safe_regex Past g* (*from_mregex mr φs*) ⟹
ms ∈ *RPD mr* ⟹ *safe_regex Past g* (*from_mregex ms φs*)
⟨*proof*⟩

lemma *RPDi_safe*: *safe_regex Past g* (*from_mregex mr φs*) ⟹
ms ∈ *RPDi n mr* ==> *safe_regex Past g* (*from_mregex ms φs*)
⟨*proof*⟩

lemma *RPDs_safe*: *safe_regex Past g* (*from_mregex mr φs*) ⟹
ms ∈ *RPDs mr* ==> *safe_regex Past g* (*from_mregex ms φs*)
⟨*proof*⟩

lemma *RPD_safe_fv_regex*: *safe_regex Past Strict* (*from_mregex mr φs*) ⟹
ms ∈ *RPD mr* ⟹ *fv_regex* (*from_mregex ms φs*) = *fv_regex* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *RPDi_safe_fv_regex*: *safe_regex Past Strict* (*from_mregex mr φs*) ⟹
ms ∈ *RPDi n mr* ==> *fv_regex* (*from_mregex ms φs*) = *fv_regex* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *RPDs_safe_fv_regex*: *safe_regex Past Strict* (*from_mregex mr φs*) ⟹
ms ∈ *RPDs mr* ==> *fv_regex* (*from_mregex ms φs*) = *fv_regex* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *RPD_ok*: *ok m mr* ⟹ *ms* ∈ *RPD mr* ⟹ *ok m ms*
⟨*proof*⟩

lemma *RPDi_ok*: *ok m mr* ⟹ *ms* ∈ *RPDi n mr* ⟹ *ok m ms*
⟨*proof*⟩

lemma *RPDs_ok*: *ok m mr* ⟹ *ms* ∈ *RPDs mr* ⟹ *ok m ms*
⟨*proof*⟩

lemma *LPD_fv_regex_le*:
ms ∈ *LPD mr* ⟹ *fv_regex* (*from_mregex ms φs*) ⊆ *fv_regex* (*from_mregex mr φs*)
⟨*proof*⟩

lemma *LPD_safe*: *safe_regex Futu g* (*from_mregex mr φs*) ⟹
ms ∈ *LPD mr* ==> *safe_regex Futu g* (*from_mregex ms φs*)
⟨*proof*⟩

lemma *LPDi_safe*: *safe_regex Futu g* (*from_mregex mr φs*) ⟹
ms ∈ *LPDi n mr* ==> *safe_regex Futu g* (*from_mregex ms φs*)
⟨*proof*⟩

lemma *LPDs_safe*: *safe_regex Futu g* (*from_mregex mr φs*) ⟹

ms ∈ *LPDs mr* ==> *safe_regex Futu g* (*from_mregex ms* φ*s*)
⟨*proof*⟩

lemma *LPD_safe_fv_regex*: *safe_regex Futu Strict* (*from_mregex mr* φ*s*) ⟹
ms ∈ *LPD mr* ==> *fv_regex* (*from_mregex ms* φ*s*) = *fv_regex* (*from_mregex mr* φ*s*)
⟨*proof*⟩

lemma *LPDi_safe_fv_regex*: *safe_regex Futu Strict* (*from_mregex mr* φ*s*) ⟹
ms ∈ *LPDi n mr* ==> *fv_regex* (*from_mregex ms* φ*s*) = *fv_regex* (*from_mregex mr* φ*s*)
⟨*proof*⟩

lemma *LPDs_safe_fv_regex*: *safe_regex Futu Strict* (*from_mregex mr* φ*s*) ⟹
ms ∈ *LPDs mr* ==> *fv_regex* (*from_mregex ms* φ*s*) = *fv_regex* (*from_mregex mr* φ*s*)
⟨*proof*⟩

lemma *LPD_ok*: *ok m mr* ⟹ *ms* ∈ *LPD mr* ⟹ *ok m ms*
⟨*proof*⟩

lemma *LPDi_ok*: *ok m mr* ⟹ *ms* ∈ *LPDi n mr* ⟹ *ok m ms*
⟨*proof*⟩

lemma *LPDs_ok*: *ok m mr* ⟹ *ms* ∈ *LPDs mr* ⟹ *ok m ms*
⟨*proof*⟩

lemma *update_matchP*:
assumes *pre*: *wf_matchP_aux* σ *V n R I r aux ne*
and *safe*: *safe_regex Past Strict r*
and *mr*: *to_mregex r* = (*mr*, φ*s*)
and *mrs*: *mrs* = *sorted_list_of_set* (*RPDs mr*)
and *qtables*: *list_all2* (λφ *rel. qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v. Formula.sat* σ *V* (*map the v*) *ne* φ) *rel*) φ*s rels*
and *result_eq*: (*rel*, *aux*′) = *update_matchP n I mr mrs rels* (τ σ *ne*) *aux*
shows *wf_matchP_aux* σ *V n R I r aux*′ (*Suc ne*)
and *qtable n* (*Formula.fv_regex r*) (*mem_restr R*) (λ*v. Formula.sat* σ *V* (*map the v*) *ne* (*Formula.MatchP I r*)) *rel*
⟨*proof*⟩

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

lemma *wf_update_until_auxlist*:
assumes *pre*: *wf_until_auxlist* σ *V n R pos* φ *I* ψ *auxlist ne*
and *qtable1*: *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v. Formula.sat* σ *V* (*map the v*) (*ne* + *length auxlist*) φ) *rel1*
and *qtable2*: *qtable n* (*Formula.fv* ψ) (*mem_restr R*) (λ*v. Formula.sat* σ *V* (*map the v*) (*ne* + *length auxlist*) 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_pos*: *args_pos args* = *pos*
shows *wf_until_auxlist* σ *V n R pos* φ *I* ψ (*update_until args rel1 rel2* (τ σ (*ne* + *length auxlist*)) *auxlist*) *ne*
⟨*proof*⟩

lemma (**in** *muaux*) *wf_update_until*:
assumes *pre*: *wf_until_aux* σ *V R args* φ ψ *aux ne*
and *qtable1*: *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v. Formula.sat* σ *V* (*map the v*) (*ne* + *length_muaux args aux*) φ) *rel1*

and *qtable2*: *qtable n* (*Formula.fv ψ*) (*mem_restr R*) (λ*v*. *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*⟩

lemma *length_update_matchF_base*:
length (*fst* (*update_matchF_base I mr mrs nt entry st*)) = *Suc 0*
⟨*proof*⟩

lemma *length_update_matchF_step*:
length (*fst* (*update_matchF_step I mr mrs nt entry st*)) = *Suc* (*length* (*fst st*))
⟨*proof*⟩

lemma *length_foldr_update_matchF_step*:
length (*fst* (*foldr* (*update_matchF_step I mr mrs nt*) *aux base*)) = *length aux* + *length* (*fst base*)
⟨*proof*⟩

lemma *length_update_matchF*: *length* (*update_matchF n I mr mrs rels nt aux*) = *Suc* (*length aux*)
⟨*proof*⟩

lemma *wf_update_matchF_base_invar*:
assumes *safe*: *safe_regex Futu Strict r*
and *mr*: *to_mregex r* = (*mr*, *φs*)
and *mrs*: *mrs* = *sorted_list_of_set* (*LPDs mr*)
and *qtables*: *list_all2* (λ*φ rel*. *qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *j φ*) *rel*) *φs rels*
shows *wf_matchF_invar σ V n R I r* (*update_matchF_base n I mr mrs rels* (*τ σ j*)) *j*
⟨*proof*⟩

lemma *Un_empty_table*[*simp*]: *rel* ∪ *empty_table* = *rel empty_table* ∪ *rel* = *rel*
⟨*proof*⟩

lemma *wf_matchF_invar_step*:
assumes *wf*: *wf_matchF_invar σ V n R I r st* (*Suc i*)
and *safe*: *safe_regex Futu Strict r*
and *mr*: *to_mregex r* = (*mr*, *φs*)
and *mrs*: *mrs* = *sorted_list_of_set* (*LPDs mr*)
and *qtables*: *list_all2* (λ*φ rel*. *qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ*) *rel*) *φs rels*
and *rel*: *qtable n* (*Formula.fv_regex r*) (*mem_restr R*) (λ*v*. (∃ *j*. *i* ≤ *j* ∧ *j* < *i* + *length* (*fst st*) ∧ *mem* (*τ σ j* − *τ σ i*) *I* ∧
Regex.match (*Formula.sat σ V* (*map the v*)) *r i j*)) *rel*
and *entry*: *entry* = (*τ σ i*, *rels*, *rel*)
and *nt*: *nt* = *τ σ* (*i* + *length* (*fst st*))
shows *wf_matchF_invar σ V n R I r* (*update_matchF_step I mr mrs nt entry st*) *i*
⟨*proof*⟩

lemma *wf_update_matchF_invar*:
assumes *pre*: *wf_matchF_aux σ V n R I r aux ne* (*length* (*fst st*) − *1*)

and *wf*: *wf_matchF_invar* σ *V n R I r st* (*ne* + *length aux*)
and *safe*: *safe_regex Futu Strict r*
and *mr*: *to_mregex r* = (*mr*, φs)
and *mrs*: *mrs* = *sorted_list_of_set* (*LPDs mr*)
and *j*: *j* = *ne* + *length aux* + *length* (*fst st*) − *1*
shows *wf_matchF_invar* σ *V n R I r* (*foldr* (*update_matchF_step I mr mrs* (τ σ *j*)) *aux st*) *ne*
⟨*proof*⟩

lemma *wf_update_matchF*:
assumes *pre*: *wf_matchF_aux* σ *V n R I r aux ne 0*
and *safe*: *safe_regex Futu Strict r*
and *mr*: *to_mregex r* = (*mr*, φs)
and *mrs*: *mrs* = *sorted_list_of_set* (*LPDs mr*)
and *nt*: *nt* = τ σ (*ne* + *length aux*)
and *qtables*: *list_all2* ($\lambda\varphi$ *rel. qtable n* (*Formula.fv* φ) (*mem_restr R*) (λv. *Formula.sat* σ *V* (*map the v*) (*ne* + *length aux*) φ) *rel*) φs *rels*
shows *wf_matchF_aux* σ *V n R I r* (*update_matchF n I mr mrs rels nt aux*) *ne 0*
⟨*proof*⟩

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

lemma *wf_matchF_aux_Cons*: *wf_matchF_aux* σ *V n R I r* (*entry* # *aux*) *ne i* ⟹
wf_matchF_aux σ *V n R I r aux* (*Suc ne*) *i*
⟨*proof*⟩

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

lemma *wf_matchF_aux_Cons1*: *wf_matchF_aux* σ *V n R I r* ((*t*, *rels*, *rel*) # *aux*) *ne i* ⟹ *t* = τ σ *ne*
⟨*proof*⟩

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

lemma *wf_matchF_aux_Cons3*: *wf_matchF_aux* σ *V n R I r* ((*t*, *rels*, *rel*) # *aux*) *ne i* ⟹
qtable n (*Formula.fv_regex r*) (*mem_restr R*) (λv. (∃ *j*. *ne* ≤ *j* ∧ *j* < *Suc* (*ne* + *length aux* + *i*) ∧ *mem* (τ σ *j* − τ σ *ne*) *I* ∧
Regex.match (*Formula.sat* σ *V* (*map the v*)) *r ne j*)) *rel*
⟨*proof*⟩

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

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

lemma *wf_mbufn_add*:
assumes *wf_mbufn i js Ps buf*
and *list_all3 list_all2 Ps* (*List.map2* (λ*j j'*. [*j*..<*j'*]) *js js'*) *xss*
and *list_all2* (≤) *js js'*
shows *wf_mbufn i js' Ps* (*mbufn_add xss buf*)
⟨*proof*⟩

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

lemma *list_all3_Nil*[*simp*]:
list_all3 P xs ys [] ⟷ *xs* = [] ∧ *ys* = []
list_all3 P xs [] *zs* ⟷ *xs* = [] ∧ *zs* = []
list_all3 P [] *ys zs* ⟷ *ys* = [] ∧ *zs* = []
⟨*proof*⟩

lemma *list_all3_Cons*:
list_all3 P xs ys (*z* # *zs*) ⟷ (∃ *x xs' y ys'*. *xs* = *x* # *xs'* ∧ *ys* = *y* # *ys'* ∧ *P x y z* ∧ *list_all3 P xs' ys' zs*)
list_all3 P xs (*y* # *ys*) *zs* ⟷ (∃ *x xs' z zs'*. *xs* = *x* # *xs'* ∧ *zs* = *z* # *zs'* ∧ *P x y z* ∧ *list_all3 P xs' ys zs'*)
list_all3 P (*x* # *xs*) *ys zs* ⟷ (∃ *y ys' z zs'*. *ys* = *y* # *ys'* ∧ *zs* = *z* # *zs'* ∧ *P x y z* ∧ *list_all3 P xs ys' zs'*)
⟨*proof*⟩

lemma *list_all3_mono_strong*: *list_all3 P xs ys zs* ⟹
(⋀*x y z*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *z* ∈ *set zs* ⟹ *P x y z* ⟹ *Q x y z*) ⟹
list_all3 Q xs ys zs
⟨*proof*⟩

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 [] ∈ *set buf* ⟹ *Mini i js* = *i*
⟨*proof*⟩

lemma *wf_mbufn_notin_set*:
assumes *wf_mbufn i js Ps buf*
shows [] ∉ *set buf* ⟹ *j* ∈ *set js* ⟹ *i* < *j*
⟨*proof*⟩

lemma *wf_mbufn_map_tl*:
wf_mbufn i js Ps buf ⟹ [] ∉ *set buf* ⟹ *wf_mbufn* (*Suc i*) *js Ps* (*map tl buf*)
⟨*proof*⟩

lemma *list_all3_list_all2I*: *list_all3* (λ*x y z*. *Q x z*) *xs ys zs* ⟹ *list_all2 Q xs zs*
⟨*proof*⟩

lemma *mbuf2t_take_eqD*:
assumes *mbuf2t_take f z buf nts* = (*z'*, *buf'*, *nts'*)
and *wf_mbuf2 i ja jb P Q buf*

and *list_all2 R* [*i*..<*j*] *nts*
and *ja* ≤ *j jb* ≤ *j*
shows *wf_mbuf2* (*min ja jb*) *ja jb P Q buf'*
and *list_all2 R* [*min ja jb*..<*j*] *nts'*
⟨*proof*⟩

lemma *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'*
⟨*proof*⟩

lemma *mbufnt_take_eqD*:
assumes *mbufnt_take f z buf nts* = (*z'*, *buf'*, *nts'*)
and *wf_mbufn i js Ps buf*
and *list_all2 R* [*i*..<*j*] *nts*
and ⋀*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'*
⟨*proof*⟩

lemma *mbuf2t_take_induct*[*consumes 5*, *case_names base step*]:
assumes *mbuf2t_take f z buf nts* = (*z'*, *buf'*, *nts'*)
and *wf_mbuf2 i ja jb P Q buf*
and *list_all2 R* [*i*..<*j*] *nts*
and *ja* ≤ *j jb* ≤ *j*
and *U i z*
and ⋀*k x y t z*. *i* ≤ *k* ⟹ *Suc k* ≤ *ja* ⟹ *Suc k* ≤ *jb* ⟹
P k x ⟹ *Q k y* ⟹ *R k t* ⟹ *U k z* ⟹ *U* (*Suc k*) (*f x y t z*)
shows *U* (*min ja jb*) *z'*
⟨*proof*⟩

lemma *list_all2_hdD*:
assumes *list_all2 P* [*i*..<*j*] *xs xs* ≠ []
shows *P i* (*hd xs*) *i* < *j*
⟨*proof*⟩

lemma *mbufn_take_induct*[*consumes 3*, *case_names base step*]:
assumes *mbufn_take f z buf* = (*z'*, *buf'*)
and *wf_mbufn i js Ps buf*
and *U i z*
and ⋀*k xs z*. *i* ≤ *k* ⟹ *Suc k* ≤ *Mini i js* ⟹
list_all2 (λ*P x*. *P k x*) *Ps xs* ⟹ *U k z* ⟹ *U* (*Suc k*) (*f xs z*)
shows *U* (*Mini i js*) *z'*
⟨*proof*⟩

lemma *mbufnt_take_induct*[*consumes 5*, *case_names base step*]:
assumes *mbufnt_take f z buf nts* = (*z'*, *buf'*, *nts'*)
and *wf_mbufn i js Ps buf*
and *list_all2 R* [*i*..<*j*] *nts*
and ⋀*k*. *k* ∈ *set js* ⟹ *k* ≤ *j*
and *U i z*
and ⋀*k xs t z*. *i* ≤ *k* ⟹ *Suc k* ≤ *Mini j js* ⟹
list_all2 (λ*P 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'*
⟨*proof*⟩

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* (λ*i*. *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* φ))
[*progress* σ *P* φ *j*..<*progress* σ *P'* φ *j'*] *xs*
and *ys*: *list_all2* (λ*i*. *qtable n* (*Formula.fv* ψ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* ψ))
[*progress* σ *P* ψ *j*..<*progress* σ *P'* ψ *j'*] *ys*
and *j* ≤ *j'*
shows *wf_mbuf2'* σ *P' V j' n R* φ ψ *buf'*
and *list_all2* (λ*i Z*. ∃ *X Y*.
qtable n (*Formula.fv* φ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* φ) *X* ∧
qtable n (*Formula.fv* ψ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* ψ) *Y* ∧
Z = *f X Y*)
[*min* (*progress* σ *P* φ *j*) (*progress* σ *P* ψ *j*)..<*min* (*progress* σ *P'* φ *j'*) (*progress* σ *P'* ψ *j'*)] *zs*
⟨*proof*⟩

lemma *mbuf2t_take_add'*:
assumes *eq*: *mbuf2t_take f z* (*mbuf2_add xs ys buf*) *nts* = (*z'*, *buf'*, *nts'*)
and *bounded*: *pred_mapping* (λ*x*. *x* ≤ *j*) *P pred_mapping* (λ*x*. *x* ≤ *j'*) *P'*
and *rm*: *rel_mapping* (≤) *P P'*
and *pre_buf*: *wf_mbuf2'* σ *P V j n R* φ ψ *buf*
and *pre_nts*: *list_all2* (λ*i t*. *t* = τ σ *i*) [*min* (*progress* σ *P* φ *j*) (*progress* σ *P* ψ *j*)..<*j'*] *nts*
and *xs*: *list_all2* (λ*i*. *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* φ))
[*progress* σ *P* φ *j*..<*progress* σ *P'* φ *j'*] *xs*
and *ys*: *list_all2* (λ*i*. *qtable n* (*Formula.fv* ψ) (*mem_restr R*) (λ*v*. *Formula.sat* σ *V* (*map the v*) *i* ψ))
[*progress* σ *P* ψ *j*..<*progress* σ *P'* ψ *j'*] *ys*
and *j* ≤ *j'*
shows *wf_mbuf2'* σ *P' V j' n R* φ ψ *buf'*
and *wf_ts* σ *P' j'* φ ψ *nts'*
⟨*proof*⟩

lemma *ok_0_atms*: *ok 0 mr* ⟹ *regex.atms* (*from_mregex mr* []) = {}
⟨*proof*⟩

lemma *ok_0_progress*: *ok 0 mr* ⟹ *progress_regex* σ *P* (*from_mregex mr* []) *j* = *j*
⟨*proof*⟩

lemma *atms_empty_atms*: *safe_regex m g r* ⟹ *atms r* = {} ⟷ *regex.atms r* = {}
⟨*proof*⟩

lemma *atms_empty_progress*: *safe_regex m g r* ⟹ *atms r* = {} ⟹ *progress_regex* σ *P r j* = *j*
⟨*proof*⟩

lemma *to_mregex_empty_progress*: *safe_regex m g r* ⟹ *to_mregex r* = (*mr*, []) ⟹
progress_regex σ *P r j* = *j*
⟨*proof*⟩

lemma *progress_regex_le*: *pred_mapping* (λ*x*. *x* ≤ *j*) *P* ⟹ *progress_regex* σ *P r j* ≤ *j*
⟨*proof*⟩

lemma *Neg_acyclic*: *formula.Neg x* = *x* ⟹ *P*
⟨*proof*⟩

lemma *case_Neg_in_iff*: *x* ∈ (*case y of formula.Neg* φ' ⇒ {φ'} | _ ⇒ {}) ⟷ *y* = *formula.Neg x*
⟨*proof*⟩

lemma *atms_nonempty_progress*:

safe_regex m g r ⟹ *atms r* ≠ {} ⟹ (λφ. *progress σ P φ j*) ‘ *atms r* = (λφ. *progress σ P φ j*) ‘ *regex.atms r*
⟨*proof*⟩

lemma *to_mregex_nonempty_progress*: *safe_regex m g r* ⟹ *to_mregex r* = (*mr*, *φs*) ⟹ *φs* ≠ [] ⟹
progress_regex σ P r j = (*MIN* φ∈*set φs*. *progress σ P φ j*)
⟨*proof*⟩

lemma *to_mregex_progress*: *safe_regex m g r* ⟹ *to_mregex r* = (*mr*, *φs*) ⟹
progress_regex σ P r j = (*if φs* = [] *then j else* (*MIN* φ∈*set φs*. *progress σ P φ j*))
⟨*proof*⟩

lemma *mbufnt_take_add'*:
assumes *eq*: *mbufnt_take f z* (*mbufn_add xss buf*) *nts* = (*z'*, *buf'*, *nts'*)
and *bounded*: *pred_mapping* (λ*x*. *x* ≤ *j*) *P pred_mapping* (λ*x*. *x* ≤ *j'*) *P'*
and *rm*: *rel_mapping* (≤) *P P'*
and *safe*: *safe_regex m g r*
and *mr*: *to_mregex r* = (*mr*, *φs*)
and *pre_buf*: *wf_mbufn' σ P V j n R r buf*
and *pre_nts*: *list_all2* (λ*i t*. *t* = *τ σ i*) [*progress_regex σ P r j*..<*j'*] *nts*
and *xss*: *list_all3 list_all2*
(*map* (λφ *i*. *qtable n* (*fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ*)) *φs*)
(*map2 upt* (*map* (λφ. *progress σ P φ j*) *φs*) (*map* (λφ. *progress σ P' φ j'*) *φs*)) *xss*
and *j* ≤ *j'*
shows *wf_mbufn' σ P' V j' n R r buf'*
and *wf_ts_regex σ P' j' r nts'*
⟨*proof*⟩

lemma *mbuf2t_take_add_induct'*[*consumes 6*, *case_names base step*]:
assumes *eq*: *mbuf2t_take f z* (*mbuf2_add xs ys buf*) *nts* = (*z'*, *buf'*, *nts'*)
and *bounded*: *pred_mapping* (λ*x*. *x* ≤ *j*) *P pred_mapping* (λ*x*. *x* ≤ *j'*) *P'*
and *rm*: *rel_mapping* (≤) *P P'*
and *pre_buf*: *wf_mbuf2' σ P V j n R φ ψ buf*
and *pre_nts*: *list_all2* (λ*i t*. *t* = *τ σ i*) [*min* (*progress σ P φ j*) (*progress σ P ψ j*)..<*j'*] *nts*
and *xs*: *list_all2* (λ*i*. *qtable n* (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ*))
[*progress σ P φ j*..<*progress σ P' φ j'*] *xs*
and *ys*: *list_all2* (λ*i*. *qtable n* (*Formula.fv ψ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i ψ*))
[*progress σ P ψ j*..<*progress σ P' ψ j'*] *ys*
and *j* ≤ *j'*
and *base*: *U* (*min* (*progress σ P φ j*) (*progress σ P ψ j*)) *z*
and *step*: ⋀*k X Y z*. *min* (*progress σ P φ j*) (*progress σ P ψ j*) ≤ *k* ⟹
Suc k ≤ *progress σ P' φ j'* ⟹ *Suc k* ≤ *progress σ P' ψ j'* ⟹
qtable n (*Formula.fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *k φ*) *X* ⟹
qtable n (*Formula.fv ψ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *k ψ*) *Y* ⟹
U k z ⟹ *U* (*Suc k*) (*f X Y* (*τ σ k*) *z*)
shows *U* (*min* (*progress σ P' φ j'*) (*progress σ P' ψ j'*)) *z'*
⟨*proof*⟩

lemma *mbufnt_take_add_induct'*[*consumes 6*, *case_names base step*]:
assumes *eq*: *mbufnt_take f z* (*mbufn_add xss buf*) *nts* = (*z'*, *buf'*, *nts'*)
and *bounded*: *pred_mapping* (λ*x*. *x* ≤ *j*) *P pred_mapping* (λ*x*. *x* ≤ *j'*) *P'*
and *rm*: *rel_mapping* (≤) *P P'*
and *safe*: *safe_regex m g r*
and *mr*: *to_mregex r* = (*mr*, *φs*)
and *pre_buf*: *wf_mbufn' σ P V j n R r buf*
and *pre_nts*: *list_all2* (λ*i t*. *t* = *τ σ i*) [*progress_regex σ P r j*..<*j'*] *nts*
and *xss*: *list_all3 list_all2*
(*map* (λφ *i*. *qtable n* (*fv φ*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ*)) *φs*)

(*map2 upt* (*map* (λφ. *progress* σ *P* φ *j*) φs) (*map* (λφ. *progress* σ *P'* φ *j'*) φs)) *xss*
and $j \leq j'$
and *base*: *U* (*progress_regex* σ *P r j*) *z*
and *step*: ⋀*k Xs z*. *progress_regex* σ *P r j* ≤ *k* ⟹ *Suc k* ≤ *progress_regex* σ *P' r j'* ⟹
list_all2 (λφ. *qtable n* (*Formula.fv* φ) (*mem_restr R*) (λv. *Formula.sat* σ *V* (*map the v*) *k* φ)) φs *Xs* ⟹
U k z ⟹ *U* (*Suc k*) (*f Xs* (τ σ *k*) *z*)
shows *U* (*progress_regex* σ *P'* *r j'*) *z'*
⟨*proof*⟩

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

lemma *progress_MatchF_le*: *progress* σ *P* (*Formula.MatchF I r*) *j* ≤ *progress_regex* σ *P r j*
⟨*proof*⟩

lemma *list_all2_upt_Cons*: *P a x* ⟹ *list_all2 P* [*Suc a*..<*b*] *xs* ⟹ *Suc a* ≤ *b* ⟹
list_all2 P [*a*..<*b*] (*x* # *xs*)
⟨*proof*⟩

lemma *list_all2_upt_append*: *list_all2 P* [*a*..<*b*] *xs* ⟹ *list_all2 P* [*b*..<*c*] *ys* ⟹
a ≤ *b* ⟹ *b* ≤ *c* ⟹ *list_all2 P* [*a*..<*c*] (*xs* @ *ys*)
⟨*proof*⟩

lemma *list_all3_list_all2_conv*: *list_all3 R xs xs ys* = *list_all2* (λx. *R x x*) *xs ys*
⟨*proof*⟩

lemma *map_split_map*: *map_split f* (*map g xs*) = *map_split* (*f o g*) *xs*
⟨*proof*⟩

lemma *map_split_alt*: *map_split f xs* = (*map* (*fst o f*) *xs*, *map* (*snd o f*) *xs*)
⟨*proof*⟩

lemma *fv_formula_of_constraint*: *fv* (*formula_of_constraint* (*t1*, *p*, *c*, *t2*)) = *fv_trm t1* ∪ *fv_trm t2*
⟨*proof*⟩

lemma (**in** *maux*) *wf_mformula_wf_set*: *wf_mformula* σ *j P V n R* φ φ' ⟹ *wf_set n* (*Formula.fv* φ')
⟨*proof*⟩

lemma *qtable_mmulti_join*:
assumes *pos*: *list_all3* (λ*A Qi X*. *qtable n A P Qi X* ∧ *wf_set n A*) *A_pos Q_pos L_pos*
and *neg*: *list_all3* (λ*A Qi X*. *qtable n A P Qi X* ∧ *wf_set n A*) *A_neg Q_neg L_neg*
and *C_eq*: *C* = ⋃(*set A_pos*) **and** *L_eq*: *L* = *L_pos* @ *L_neg*
and *A_pos* ≠ [] **and** *fv_subset*: ⋃(*set A_neg*) ⊆ ⋃(*set A_pos*)
and *restrict_pos*: ⋀*x*. *wf_tuple n C x* ⟹ *P x* ⟹ *list_all* (λ*A*. *P* (*restrict A x*)) *A_pos*
and *restrict_neg*: ⋀*x*. *wf_tuple n C x* ⟹ *P x* ⟹ *list_all* (λ*A*. *P* (*restrict A x*)) *A_neg*
and *Qs*: ⋀*x*. *wf_tuple n C x* ⟹ *P x* ⟹ *Q x* ⟷
list_all2 (λ*A Qi*. *Qi* (*restrict A x*)) *A_pos Q_pos* ∧
list_all2 (λ*A Qi*. ¬ *Qi* (*restrict A x*)) *A_neg Q_neg*
shows *qtable n C P Q* (*mmulti_join n A_pos A_neg L*)
⟨*proof*⟩

lemma *nth_filter*: *i* < *length* (*filter P xs*) ⟹
(⋀*i'*. *i'* < *length xs* ⟹ *P* (*xs* ! *i'*) ⟹ *Q* (*xs* ! *i'*)) ⟹ *Q* (*filter P xs* ! *i*)
⟨*proof*⟩

lemma *nth_partition*: *i* < *length xs* ⟹

$(\bigwedge i'.\ i' < \mathit{length}\ (\mathit{filter}\ P\ xs) \Longrightarrow Q\ (\mathit{filter}\ P\ xs\ !\ i')) \Longrightarrow$
$(\bigwedge i'.\ i' < \mathit{length}\ (\mathit{filter}\ (\mathit{Not} \circ P)\ xs) \Longrightarrow Q\ (\mathit{filter}\ (\mathit{Not} \circ P)\ xs\ !\ i')) \Longrightarrow Q\ (xs\ !\ i)$
⟨*proof*⟩

lemma *qtable_bin_join*:
assumes *qtable n A P Q1 X qtable n B P Q2 Y* ¬ *b* ⟹ $B \subseteq A$ $C = A \cup B$
$\bigwedge x.$ *wf_tuple n C x* ⟹ *P x* ⟹ *P* (*restrict A x*) ∧ *P* (*restrict B x*)
$\bigwedge x.$ *b* ⟹ *wf_tuple n C x* ⟹ *P x* ⟹ *Q x* ⟷ *Q1* (*restrict A x*) ∧ *Q2* (*restrict B x*)
$\bigwedge x.$ ¬ *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*)
⟨*proof*⟩

lemma *restrict_update*: $y \notin A \Longrightarrow y < \mathit{length}\ x \Longrightarrow \mathit{restrict}\ A\ (x[y:=z]) = \mathit{restrict}\ A\ x$
⟨*proof*⟩

lemma *qtable_assign*:
assumes *qtable n A P Q X*
$y < n$ *insert y A* = *A'* $y \notin A$
$\bigwedge x'.$ *wf_tuple n A' x'* ⟹ *P x'* ⟹ *P* (*restrict A x'*)
$\bigwedge x.$ *wf_tuple n A x* ⟹ *P x* ⟹ *Q x* ⟹ *Q'* ($x[y:=\mathit{Some}\ (f\ x)]$)
$\bigwedge x'.$ *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'* ((λ*x*. $x[y:=\mathit{Some}\ (f\ x)]$) ‘ *X*) (**is** *qtable _ _ _ _ ?Y*)
⟨*proof*⟩

lemma *sat_the_update*: $y \notin \mathit{fv}\ \varphi \Longrightarrow \mathit{Formula.sat}\ \sigma\ V\ (\mathit{map}\ \mathit{the}\ (x[y:=z]))\ i\ \varphi = \mathit{Formula.sat}\ \sigma\ V\ (\mathit{map}\ \mathit{the}\ x)\ i\ \varphi$
⟨*proof*⟩

lemma *progress_constraint*: *progress* σ *P* (*formula_of_constraint c*) *j* = *j*
⟨*proof*⟩

lemma *qtable_filter*:
assumes *qtable n A P Q X*
$\bigwedge x.$ *wf_tuple n A x* ⟹ *P x* ⟹ *Q x* ∧ *R x* ⟷ *Q' x*
shows *qtable n A P Q'* (*Set.filter R X*) (**is** *qtable _ _ _ _ ?Y*)
⟨*proof*⟩

lemma *eval_constraint_sat_eq*: *wf_tuple n A x* ⟹ *fv_trm t1* ⊆ *A* ⟹ *fv_trm t2* ⊆ *A* ⟹
∀ *i*∈*A*. *i* < *n* ⟹ *eval_constraint* (*t1*, *p*, *c*, *t2*) *x* =
Formula.sat σ *V* (*map the x*) *i* (*formula_of_constraint* (*t1*, *p*, *c*, *t2*))
⟨*proof*⟩

declare *progress_le_gen*[*simp*]

definition *wf_envs* σ *j P P' V db* =
(*dom V* = *dom P* ∧
Mapping.keys db = *dom P* ∪ {*p*. *p* ∈ *fst* ‘ Γ σ *j*} ∧
rel_mapping (≤) *P P'* ∧
pred_mapping (λ*i*. *i* ≤ *j*) *P* ∧
pred_mapping (λ*i*. *i* ≤ *Suc j*) *P'* ∧
(∀ *p* ∈ *Mapping.keys db* − *dom P*. *the* (*Mapping.lookup db p*) = [{*ts*. (*p*, *ts*) ∈ Γ σ *j*}]) ∧
(∀ *p* ∈ *dom P*. *list_all2* (λ*i X*. *X* = *the* (*V p*) *i*) [*the* (*P p*)..<*the* (*P' p*)] (*the* (*Mapping.lookup db p*))))

lift_definition *mk_db* :: (*Formula.name* × *event_data list*) *set* ⇒ *Formula.database* **is**
λ*X p*. (*if p* ∈ *fst* ‘ *X then Some* [{*ts*. (*p*, *ts*) ∈ *X*}] *else None*) ⟨*proof*⟩

lemma *wf_envs_mk_db*: *wf_envs* σ *j Map.empty Map.empty Map.empty* (*mk_db* (Γ σ *j*))

⟨*proof*⟩

lemma *wf_envs_update*:
assumes *wf_envs*: *wf_envs σ j P P′ V db*
and *m_eq*: *m* = *Formula.nfv φ*
and *in_fv*: {*0 ..< m*} ⊆ *fv φ*
and *xs*: *list_all2* (λ*i*. *qtable m* (*Formula.fv φ*) (*mem_restr UNIV*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ*))
[*progress σ P φ j..<progress σ P′ φ* (*Suc j*)] *xs*
shows *wf_envs σ j* (*P*(*p* ↦ *progress σ P φ j*)) (*P′*(*p* ↦ *progress σ P′ φ* (*Suc j*)))
(*V*(*p* ↦ λ*i*. {*v*. *length v* = *m* ∧ *Formula.sat σ V v i φ*}))
(*Mapping.update p* (*map* (*image* (*map the*)) *xs*) *db*)
⟨*proof*⟩

lemma *wf_envs_P_simps*[*simp*]:
wf_envs σ j P P′ V db ⟹ *pred_mapping* (λ*i*. *i* ≤ *j*) *P*
wf_envs σ j P P′ V db ⟹ *pred_mapping* (λ*i*. *i* ≤ *Suc j*) *P′*
wf_envs σ j P P′ V db ⟹ *rel_mapping* (≤) *P P′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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 (λ*i*. *qtable n* (*Formula.fv φ′*) (*mem_restr R*) (λ*v*. *Formula.sat σ V* (*map the v*) *i φ′*))
[*progress σ P φ′ j..<progress σ P′ φ′* (*Suc j*)] *xs*
⟨*proof*⟩

6.6.4 Monitor step

lemma (**in** *maux*) *wf_mstate_mstep*: *wf_mstate φ π R st* ⟹ *last_ts π* ≤ *snd tdb* ⟹
wf_mstate φ (*psnoc π tdb*) *R* (*snd* (*mstep* (*map_prod mk_db id tdb*) *st*))
⟨*proof*⟩

definition *flatten_verdicts Vs* = (⋃ (*set* (*map* (λ(*i*, *X*). (λ*v*. (*i*, *v*)) ‘ *X*) *Vs*)))

lemma *flatten_verdicts_append*[*simp*]:
flatten_verdicts (*Vs* @ *Us*) = *flatten_verdicts Vs* ∪ *flatten_verdicts Us*
⟨*proof*⟩

lemma (**in** *maux*) *mstep_output_iff*:
assumes *wf_mstate φ π R st last_ts π* ≤ *snd tdb prefix_of* (*psnoc π tdb*) *σ mem_restr R v*

shows $(i, v) \in$ *flatten_verdicts* (*fst* (*mstep* (*map_prod mk_db id tdb*) *st*)) $\longleftrightarrow$
progress σ *Map.empty* φ (*plen* π) $\leq i \wedge i <$ *progress* σ *Map.empty* φ (*Suc* (*plen* π)) $\wedge$
wf_tuple (*Formula.nfv* φ) (*Formula.fv* φ) $v \wedge$ *Formula.sat* σ *Map.empty* (*map the* v) i φ
⟨*proof*⟩

6.6.5 Monitor function

locale *verimon* = *verimon_spec* + *maux*

lemma (**in** *verimon*) *mstep_mverdicts*:
assumes *wf*: *wf_mstate* φ π *R st*
and *le*[*simp*]: *last_ts* $\pi \leq$ *snd tdb*
and *restrict*: *mem_restr R v*
shows $(i, v) \in$ *flatten_verdicts* (*fst* (*mstep* (*map_prod mk_db id tdb*) *st*)) $\longleftrightarrow$
$(i, v) \in M$ (*psnoc* π *tdb*) $-$ M π
⟨*proof*⟩

context *maux*
begin

primrec *msteps0* **where**
msteps0 [] *st* = ([], *st*)
| *msteps0* (*tdb* # π) *st* =
(*let* (V', *st'*) = *mstep* (*map_prod mk_db id tdb*) *st*; (V'', *st''*) = *msteps0* π *st'* *in* (V' @ V'', *st''*))

primrec *msteps0_stateless* **where**
msteps0_stateless [] *st* = []
| *msteps0_stateless* (*tdb* # π) *st* = (*let* (V', *st'*) = *mstep* (*map_prod mk_db id tdb*) *st* *in* V' @ *msteps0_stateless* π *st'*)

lemma *msteps0_msteps0_stateless*: *fst* (*msteps0 w st*) = *msteps0_stateless w st*
⟨*proof*⟩

lift_definition *msteps* :: *Formula.prefix* ⇒ ('*msaux*, '*muaux*) *mstate* ⇒ (*nat* × *event_data table*) *list* × ('*msaux*, '*muaux*) *mstate*
is *msteps0* ⟨*proof*⟩

lift_definition *msteps_stateless* :: *Formula.prefix* ⇒ ('*msaux*, '*muaux*) *mstate* ⇒ (*nat* × *event_data table*) *list*
is *msteps0_stateless* ⟨*proof*⟩

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

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

lemma *msteps_psnoc*: *last_ts* $\pi \leq$ *snd tdb* $\Longrightarrow$ *msteps* (*psnoc* π *tdb*) *st* =
(*let* (V', *st'*) = *msteps* π *st*; (V'', *st''*) = *mstep* (*map_prod mk_db id tdb*) *st'* *in* (V' @ V'', *st''*))
⟨*proof*⟩

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

end

lemma *Suc_length_conv_snoc*: (*Suc n* = *length xs*) = ($\exists$ *y ys*. *xs* = *ys* @ [*y*] $\wedge$ *length ys* = *n*)
⟨*proof*⟩

lemma (**in** *verimon*) *wf_mstate_msteps*: *wf_mstate* $\varphi\ \pi\ R\ st \Longrightarrow$ *mem_restr* $R\ v \Longrightarrow \pi \leq \pi' \Longrightarrow$
$X =$ *msteps* (*pdrop* (*plen* π) π') $st \Longrightarrow$ *wf_mstate* $\varphi\ \pi'\ R$ (*snd* X) $\wedge$
$((i, v) \in$ *flatten_verdicts* (*fst* X)) $= ((i, v) \in M\ \pi' - M\ \pi)$
⟨*proof*⟩

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* π) π') st) $\longleftrightarrow (i, v) \in M\ \pi' - M\ \pi$
⟨*proof*⟩

lemma (**in** *verimon*) *wf_mstate_msteps_stateless_UNIV*: *wf_mstate* $\varphi\ \pi$ *UNIV* $st \Longrightarrow \pi \leq \pi' \Longrightarrow$
flatten_verdicts (*msteps_stateless* (*pdrop* (*plen* π) π') st) $= M\ \pi' - M\ \pi$
⟨*proof*⟩

lemma (**in** *verimon*) *mverdicts_Nil*: M *pnil* $= \{\}$
⟨*proof*⟩

context *maux*
begin

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

lemma *wf_mstate_minit_safe*: *mmonitorable* $\varphi \Longrightarrow$ *wf_mstate* φ *pnil* R (*minit_safe* φ)
⟨*proof*⟩

end

lemma (**in** *verimon*) *monitor_mverdicts*: *flatten_verdicts* (*monitor* $\varphi\ \pi$) $= M\ \pi$
⟨*proof*⟩

6.7 Collected correctness results

context *verimon*
begin

We summarize the main results proved above.

1. The term M describes semantically the monitor's expected behaviour:
   - *mono_monitor*: $\pi \leq \pi' \Longrightarrow M\ \pi \subseteq M\ \pi'$
   - *sound_monitor*: $[\![(i, v) \in M\ \pi;$ *prefix_of* $\pi\ \sigma]\!] \Longrightarrow$ *Formula.sat* σ *Map.empty* (*map the* v) $i\ \varphi$
   - *complete_monitor*: $[\![$*prefix_of* $\pi\ \sigma$; *wf_tuple* (*Formula.nfv* φ) (*fv* φ) v; $\bigwedge\sigma$. *prefix_of* $\pi\ \sigma \Longrightarrow$ *Formula.sat* σ *Map.empty* (*map the* v) $i\ \varphi]\!] \Longrightarrow \exists\pi'$. *prefix_of* $\pi'\ \sigma \wedge (i, v) \in M\ \pi'$
   - *sliceable_M*: *mem_restr* $S\ v \Longrightarrow ((i, v) \in M$ (*pmap_*Γ (λD. $D \cap$ *relevant_events* φ S) π)) $= ((i, v) \in M\ \pi)$
2. The executable monitor's online interface *minit_safe* and *mstep* preserves the invariant *wf_mstate* and produces the the verdicts according to M:
   - *wf_mstate_minit_safe*: *mmonitorable* $\varphi' \Longrightarrow$ *wf_mstate* φ' *pnil* R (*minit_safe* φ')
   - *wf_mstate_mstep*: $[\![$*wf_mstate* $\varphi'\ \pi\ R\ st$; *last_ts* $\pi \leq$ *snd tdb*$]\!] \Longrightarrow$ *wf_mstate* φ' (*psnoc* π *tdb*) R (*snd* (*mstep* (*map_prod mk_db id tdb*) st))

- *mstep_mverdicts*: ⟦*wf_mstate φ π R st*; *last_ts π ≤ snd tdb*; *mem_restr R v*⟧ ⟹ ((*i*, *v*) ∈ *flatten_verdicts* (*fst* (*mstep* (*map_prod mk_db id tdb*) *st*))) = ((*i*, *v*) ∈ *M* (*psnoc π tdb*) − *M π*)

3. The executable monitor's offline interface *local.monitor* implements *M*:

   - *monitor_mverdicts*: *flatten_verdicts* (*local.monitor φ π*) = *M π*

end

7 Efficient implementation of temporal operators

7.1 Optimized queue data structure

lemma *less_enat_iff*: *a* < *enat i* ⟷ (∃*j*. *a* = *enat j* ∧ *j* < *i*)
⟨*proof*⟩

type_synonym *'a queue_t* = *'a list* × *'a list*

definition *queue_invariant* :: *'a queue_t* ⇒ *bool* **where**
queue_invariant q = (*case q of* ([], []) ⇒ *True* | (*fs*, *l* # *ls*) ⇒ *True* | _ ⇒ *False*)

typedef *'a queue* = {*q* :: *'a queue_t*. *queue_invariant q*}
⟨*proof*⟩

setup_lifting *type_definition_queue*

lift_definition *linearize* :: *'a queue* ⇒ *'a list* **is** (λ*q*. *case q of* (*fs*, *ls*) ⇒ *fs* @ *rev ls*) ⟨*proof*⟩

lift_definition *empty_queue* :: *'a queue* **is** ([], [])
⟨*proof*⟩

lemma *empty_queue_rep*: *linearize empty_queue* = []
⟨*proof*⟩

lift_definition *is_empty* :: *'a queue* ⇒ *bool* **is** λ*q*. (*case q of* ([], []) ⇒ *True* | _ ⇒ *False*) ⟨*proof*⟩

lemma *linearize_t_Nil*: (*case q of* (*fs*, *ls*) ⇒ *fs* @ *rev ls*) = [] ⟷ *q* = ([], [])
⟨*proof*⟩

lemma *is_empty_alt*: *is_empty q* ⟷ *linearize q* = []
⟨*proof*⟩

fun *prepend_queue_t* :: *'a* ⇒ *'a queue_t* ⇒ *'a queue_t* **where**
prepend_queue_t a ([], []) = ([], [*a*])
| *prepend_queue_t a* (*fs*, *l* # *ls*) = (*a* # *fs*, *l* # *ls*)
| *prepend_queue_t a* (*f* # *fs*, []) = *undefined*

lift_definition *prepend_queue* :: *'a* ⇒ *'a queue* ⇒ *'a queue* **is** *prepend_queue_t*
⟨*proof*⟩

lemma *prepend_queue_rep*: *linearize* (*prepend_queue a q*) = *a* # *linearize q*
⟨*proof*⟩

lift_definition *append_queue* :: *'a* ⇒ *'a queue* ⇒ *'a queue* **is**
(λ*a q*. *case q of* (*fs*, *ls*) ⇒ (*fs*, *a* # *ls*))
⟨*proof*⟩

lemma *append_queue_rep*: *linearize* (*append_queue a q*) = *linearize q* @ [*a*]
⟨*proof*⟩

fun *safe_last_t* :: *'a queue_t* ⇒ *'a option* × *'a queue_t* **where**
safe_last_t ([], []) = (*None*, ([], []))
| *safe_last_t* (*fs*, *l* # *ls*) = (*Some l*, (*fs*, *l* # *ls*))
| *safe_last_t* (*f* # *fs*, []) = *undefined*

lift_definition *safe_last* :: *'a queue* ⇒ *'a option* × *'a queue* **is** *safe_last_t*
⟨*proof*⟩

lemma *safe_last_rep*: *safe_last q* = (*α*, *q'*) ⟹ *linearize q* = *linearize q'* ∧
(*case α of None* ⇒ *linearize q* = [] | *Some a* ⇒ *linearize q* ≠ [] ∧ *a* = *last* (*linearize q*))
⟨*proof*⟩

fun *safe_hd_t* :: *'a queue_t* ⇒ *'a option* × *'a queue_t* **where**
safe_hd_t ([], []) = (*None*, ([], []))
| *safe_hd_t* ([], [*l*]) = (*Some l*, ([], [*l*]))
| *safe_hd_t* ([], *l* # *ls*) = (*let fs* = *rev ls in* (*Some* (*hd fs*), (*fs*, [*l*])))
| *safe_hd_t* (*f* # *fs*, *l* # *ls*) = (*Some f*, (*f* # *fs*, *l* # *ls*))
| *safe_hd_t* (*f* # *fs*, []) = *undefined*

lift_definition(*code_dt*) *safe_hd* :: *'a queue* ⇒ *'a option* × *'a queue* **is** *safe_hd_t*
⟨*proof*⟩

lemma *safe_hd_rep*: *safe_hd q* = (*α*, *q'*) ⟹ *linearize q* = *linearize q'* ∧
(*case α of None* ⇒ *linearize q* = [] | *Some a* ⇒ *linearize q* ≠ [] ∧ *a* = *hd* (*linearize q*))
⟨*proof*⟩

fun *replace_hd_t* :: *'a* ⇒ *'a queue_t* ⇒ *'a queue_t* **where**
replace_hd_t a ([], []) = ([], [])
| *replace_hd_t a* ([], [*l*]) = ([], [*a*])
| *replace_hd_t a* ([], *l* # *ls*) = (*let fs* = *rev ls in* (*a* # *tl fs*, [*l*]))
| *replace_hd_t a* (*f* # *fs*, *l* # *ls*) = (*a* # *fs*, *l* # *ls*)
| *replace_hd_t a* (*f* # *fs*, []) = *undefined*

lift_definition *replace_hd* :: *'a* ⇒ *'a queue* ⇒ *'a queue* **is** *replace_hd_t*
⟨*proof*⟩

lemma *tl_append*: *xs* ≠ [] ⟹ *tl xs* @ *ys* = *tl* (*xs* @ *ys*)
⟨*proof*⟩

lemma *replace_hd_rep*: *linearize q* = *f* # *fs* ⟹ *linearize* (*replace_hd a q*) = *a* # *fs*
⟨*proof*⟩

fun *replace_last_t* :: *'a* ⇒ *'a queue_t* ⇒ *'a queue_t* **where**
replace_last_t a ([], []) = ([], [])
| *replace_last_t a* (*fs*, *l* # *ls*) = (*fs*, *a* # *ls*)
| *replace_last_t a* (*fs*, []) = *undefined*

lift_definition *replace_last* :: *'a* ⇒ *'a queue* ⇒ *'a queue* **is** *replace_last_t*
⟨*proof*⟩

lemma *replace_last_rep*: *linearize q* = *fs* @ [*f*] ⟹ *linearize* (*replace_last a q*) = *fs* @ [*a*]
⟨*proof*⟩

fun *tl_queue_t* :: *'a queue_t* ⇒ *'a queue_t* **where**

```
  tl_queue_t ([], []) = ([], [])
| tl_queue_t ([], [l]) = ([], [])
| tl_queue_t ([], l # ls) = (tl (rev ls), [l])
| tl_queue_t (a # as, fs) = (as, fs)

lift_definition tl_queue :: 'a queue ⇒ 'a queue is tl_queue_t
  ⟨proof⟩

lemma tl_queue_rep: ¬is_empty q ⟹ linearize (tl_queue q) = tl (linearize q)
  ⟨proof⟩

lemma length_tl_queue_rep: ¬is_empty q ⟹
  length (linearize (tl_queue q)) < length (linearize q)
  ⟨proof⟩

lemma length_tl_queue_safe_hd:
  assumes safe_hd q = (Some a, q')
  shows length (linearize (tl_queue q')) < length (linearize q)
  ⟨proof⟩

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')
  ⟨proof⟩
termination
  ⟨proof⟩

lemma dropWhile_hd_tl: xs ≠ [] ⟹
  dropWhile P xs = (if P (hd xs) then dropWhile P (tl xs) else xs)
  ⟨proof⟩

lemma dropWhile_queue_rep: linearize (dropWhile_queue f q) = dropWhile f (linearize q)
  ⟨proof⟩

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)
  ⟨proof⟩
termination
  ⟨proof⟩

lemma takeWhile_hd_tl: xs ≠ [] ⟹
  takeWhile P xs = (if P (hd xs) then hd xs # takeWhile P (tl xs) else [])
  ⟨proof⟩

lemma takeWhile_queue_rep: linearize (takeWhile_queue f q) = takeWhile f (linearize q)
  ⟨proof⟩

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', []))
  ⟨proof⟩
termination
  ⟨proof⟩
```

lemma *takedropWhile_queue_fst*: *fst* (*takedropWhile_queue f q*) = *dropWhile_queue f q*
⟨*proof*⟩

lemma *takedropWhile_queue_snd*: *snd* (*takedropWhile_queue f q*) = *takeWhile f* (*linearize q*)
⟨*proof*⟩

7.2 Optimized data structure for Since

type_synonym *'a mmsaux* = *ts* × *ts* × *bool list* × *bool list* ×
(*ts* × *'a table*) *queue* × (*ts* × *'a table*) *queue* ×
((*'a tuple*, *ts*) *mapping*) × ((*'a tuple*, *ts*) *mapping*)

fun *time_mmsaux* :: *'a mmsaux* ⇒ *ts* **where**
time_mmsaux aux = (*case aux of* (*nt*, _) ⇒ *nt*)

definition *ts_tuple_rel* :: (*ts* × *'a table*) *set* ⇒ (*ts* × *'a tuple*) *set* **where**
ts_tuple_rel ys = {(*t*, *as*). ∃ *X*. *as* ∈ *X* ∧ (*t*, *X*) ∈ *ys*}

lemma *finite_fst_ts_tuple_rel*: *finite* (*fst* ‘ {*tas* ∈ *ts_tuple_rel* (*set xs*). *P tas*})
⟨*proof*⟩

lemma *ts_tuple_rel_ext_Cons*: *tas* ∈ *ts_tuple_rel* {(*nt*, *X*)} ⟹
tas ∈ *ts_tuple_rel* (*set* ((*nt*, *X*) # *tass*))
⟨*proof*⟩

lemma *ts_tuple_rel_ext_Cons'*: *tas* ∈ *ts_tuple_rel* (*set tass*) ⟹
tas ∈ *ts_tuple_rel* (*set* ((*nt*, *X*) # *tass*))
⟨*proof*⟩

lemma *ts_tuple_rel_intro*: *as* ∈ *X* ⟹ (*t*, *X*) ∈ *ys* ⟹ (*t*, *as*) ∈ *ts_tuple_rel ys*
⟨*proof*⟩

lemma *ts_tuple_rel_dest*: (*t*, *as*) ∈ *ts_tuple_rel ys* ⟹ ∃ *X*. (*t*, *X*) ∈ *ys* ∧ *as* ∈ *X*
⟨*proof*⟩

lemma *ts_tuple_rel_Un*: *ts_tuple_rel* (*ys* ∪ *zs*) = *ts_tuple_rel ys* ∪ *ts_tuple_rel zs*
⟨*proof*⟩

lemma *ts_tuple_rel_ext*: *tas* ∈ *ts_tuple_rel* {(*nt*, *X*)} ⟹
tas ∈ *ts_tuple_rel* (*set* ((*nt*, *Y* ∪ *X*) # *tass*))
⟨*proof*⟩

lemma *ts_tuple_rel_ext'*: *tas* ∈ *ts_tuple_rel* (*set* ((*nt*, *X*) # *tass*)) ⟹
tas ∈ *ts_tuple_rel* (*set* ((*nt*, *X* ∪ *Y*) # *tass*))
⟨*proof*⟩

lemma *ts_tuple_rel_mono*: *ys* ⊆ *zs* ⟹ *ts_tuple_rel ys* ⊆ *ts_tuple_rel zs*
⟨*proof*⟩

lemma *ts_tuple_rel_filter*: *ts_tuple_rel* (*set* (*filter* (λ(*t*, *X*). *P t*) *xs*)) =
{(*t*, *X*) ∈ *ts_tuple_rel* (*set xs*). *P t*}
⟨*proof*⟩

lemma *ts_tuple_rel_set_filter*: *x* ∈ *ts_tuple_rel* (*set* (*filter P xs*)) ⟹
x ∈ *ts_tuple_rel* (*set xs*)
⟨*proof*⟩

definition *valid_tuple* :: (('*a* tuple, *ts*) *mapping*) ⇒ (*ts* × '*a* tuple) ⇒ *bool* **where**
valid_tuple tuple_since = (λ(*t*, *as*). *case Mapping.lookup tuple_since as of None* ⇒ *False*
| *Some t'* ⇒ *t* ≥ *t'*)

definition *safe_max* :: '*a* :: *linorder set* ⇒ '*a option* **where**
safe_max X = (*if X* = {} *then None else Some* (*Max X*))

lemma *safe_max_empty*: *safe_max X* = *None* ⟷ *X* = {}
⟨*proof*⟩

lemma *safe_max_empty_dest*: *safe_max X* = *None* ⟹ *X* = {}
⟨*proof*⟩

lemma *safe_max_Some_intro*: *x* ∈ *X* ⟹ ∃ *y*. *safe_max X* = *Some y*
⟨*proof*⟩

lemma *safe_max_Some_dest_in*: *finite X* ⟹ *safe_max X* = *Some x* ⟹ *x* ∈ *X*
⟨*proof*⟩

lemma *safe_max_Some_dest_le*: *finite X* ⟹ *safe_max X* = *Some x* ⟹ *y* ∈ *X* ⟹ *y* ≤ *x*
⟨*proof*⟩

fun *valid_mmsaux* :: *args* ⇒ *ts* ⇒ '*a mmsaux* ⇒ '*a Monitor.msaux* ⇒ *bool* **where**
valid_mmsaux args cur (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *ys* ⟷
(*args_L args*) ⊆ (*args_R args*) ∧
maskL = *join_mask* (*args_n args*) (*args_L args*) ∧
maskR = *join_mask* (*args_n args*) (*args_R args*) ∧
(∀ (*t*, *X*) ∈ *set ys*. *table* (*args_n args*) (*args_R args*) *X*) ∧
table (*args_n args*) (*args_R args*) (*Mapping.keys tuple_in*) ∧
table (*args_n args*) (*args_R args*) (*Mapping.keys tuple_since*) ∧
(∀ *as* ∈ ⋃(*snd* ' (*set* (*linearize data_prev*))). *wf_tuple* (*args_n args*) (*args_R args*) *as*) ∧
cur = *nt* ∧
ts_tuple_rel (*set ys*) =
{*tas* ∈ *ts_tuple_rel* (*set* (*linearize data_prev*) ∪ *set* (*linearize data_in*)).
valid_tuple tuple_since tas} ∧
sorted (*map fst* (*linearize data_prev*)) ∧
(∀ *t* ∈ *fst* ' *set* (*linearize data_prev*). *t* ≤ *nt* ∧ *nt* − *t* < *left* (*args_ivl args*)) ∧
sorted (*map fst* (*linearize data_in*)) ∧
(∀ *t* ∈ *fst* ' *set* (*linearize data_in*). *t* ≤ *nt* ∧ *nt* − *t* ≥ *left* (*args_ivl args*)) ∧
(∀ *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*})) ∧
(∀ *as* ∈ *Mapping.keys tuple_since*. *case Mapping.lookup tuple_since as of Some t* ⇒ *t* ≤ *nt*)

lemma *Mapping_lookup_filter_keys*: *k* ∈ *Mapping.keys* (*Mapping.filter f m*) ⟹
Mapping.lookup (*Mapping.filter f m*) *k* = *Mapping.lookup m k*
⟨*proof*⟩

lemma *Mapping_filter_keys*: (∀ *k* ∈ *Mapping.keys m*. *P* (*Mapping.lookup m k*)) ⟹
(∀ *k* ∈ *Mapping.keys* (*Mapping.filter f m*). *P* (*Mapping.lookup* (*Mapping.filter f m*) *k*))
⟨*proof*⟩

lemma *Mapping_filter_keys_le*: (⋀*x*. *P x* ⟹ *P' x*) ⟹
(∀ *k* ∈ *Mapping.keys m*. *P* (*Mapping.lookup m k*)) ⟹ (∀ *k* ∈ *Mapping.keys m*. *P'* (*Mapping.lookup m k*))
⟨*proof*⟩

lemma *Mapping_keys_dest*: *x* ∈ *Mapping.keys f* ⟹ ∃ *y*. *Mapping.lookup f x* = *Some y*
⟨*proof*⟩

lemma *Mapping_keys_intro*: *Mapping.lookup f x* $\neq$ *None* $\Longrightarrow$ *x* $\in$ *Mapping.keys f*
⟨*proof*⟩

lemma *valid_mmsaux_tuple_in_keys*: *valid_mmsaux args cur*
(*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *ys* $\Longrightarrow$
Mapping.keys tuple_in = *snd* ‘ {*tas* $\in$ *ts_tuple_rel* (*set* (*linearize data_in*)).
valid_tuple tuple_since tas}
⟨*proof*⟩

fun *init_mmsaux* :: *args* $\Rightarrow$ *'a mmsaux* **where**
init_mmsaux args = (*0*, *0*, *join_mask* (*args_n args*) (*args_L args*),
join_mask (*args_n args*) (*args_R args*), *empty_queue*, *empty_queue*, *Mapping.empty*, *Mapping.empty*)

lemma *valid_init_mmsaux*: *L* $\subseteq$ *R* $\Longrightarrow$ *valid_mmsaux* (*init_args I n L R b*) *0*
(*init_mmsaux* (*init_args I n L R b*)) []
⟨*proof*⟩

abbreviation *filter_cond X' ts t'* $\equiv$ (λ*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 (λ(*t*, *X*). *enat* (*nt* $-$ *t*) > *c*) *xs* = *filter* (λ(*t*, *X*). *enat* (*nt* $-$ *t*) $\leq$ *c*) *xs*
⟨*proof*⟩

lemma *dropWhile_filter'*:
fixes *nt* :: *nat*
shows *sorted* (*map fst xs*) $\Longrightarrow$ $\forall$ *t* $\in$ *fst* ‘ *set xs*. *t* $\leq$ *nt* $\Longrightarrow$
dropWhile (λ(*t*, *X*). *nt* $-$ *t* $\geq$ *c*) *xs* = *filter* (λ(*t*, *X*). *nt* $-$ *t* < *c*) *xs*
⟨*proof*⟩

lemma *dropWhile_filter''*:
sorted xs $\Longrightarrow$ $\forall$ *t* $\in$ *set xs*. *t* $\leq$ *nt* $\Longrightarrow$
dropWhile (λ*t*. *enat* (*nt* $-$ *t*) > *c*) *xs* = *filter* (λ*t*. *enat* (*nt* $-$ *t*) $\leq$ *c*) *xs*
⟨*proof*⟩

lemma *takeWhile_filter*:
sorted (*map fst xs*) $\Longrightarrow$ $\forall$ *t* $\in$ *fst* ‘ *set xs*. *t* $\leq$ *nt* $\Longrightarrow$
takeWhile (λ(*t*, *X*). *enat* (*nt* $-$ *t*) > *c*) *xs* = *filter* (λ(*t*, *X*). *enat* (*nt* $-$ *t*) > *c*) *xs*
⟨*proof*⟩

lemma *takeWhile_filter'*:
fixes *nt* :: *nat*
shows *sorted* (*map fst xs*) $\Longrightarrow$ $\forall$ *t* $\in$ *fst* ‘ *set xs*. *t* $\leq$ *nt* $\Longrightarrow$
takeWhile (λ(*t*, *X*). *nt* $-$ *t* $\geq$ *c*) *xs* = *filter* (λ(*t*, *X*). *nt* $-$ *t* $\geq$ *c*) *xs*
⟨*proof*⟩

lemma *takeWhile_filter''*:
sorted xs $\Longrightarrow$ $\forall$ *t* $\in$ *set xs*. *t* $\leq$ *nt* $\Longrightarrow$
takeWhile (λ*t*. *enat* (*nt* $-$ *t*) > *c*) *xs* = *filter* (λ*t*. *enat* (*nt* $-$ *t*) > *c*) *xs*
⟨*proof*⟩

lemma *fold_Mapping_filter_None*: *Mapping.lookup ts as* = *None* $\Longrightarrow$
Mapping.lookup (*fold* (λ(*t*, *X*) *ts*. *Mapping.filter*
(*filter_cond X ts t*) *ts*) *ds ts*) *as* = *None*
⟨*proof*⟩

lemma *Mapping_lookup_filter_Some_P*: *Mapping.lookup* (*Mapping.filter P m*) *k* = *Some v* $\Longrightarrow$ *P k v*

⟨*proof*⟩

lemma *Mapping_lookup_filter_None*: $(\bigwedge v.\ \neg P\ k\ v) \Longrightarrow$
Mapping.lookup (*Mapping.filter P m*) *k* = *None*
⟨*proof*⟩

lemma *Mapping_lookup_filter_Some*: $(\bigwedge v.\ P\ k\ v) \Longrightarrow$
Mapping.lookup (*Mapping.filter P m*) *k* = *Mapping.lookup m k*
⟨*proof*⟩

lemma *Mapping_lookup_filter_not_None*: *Mapping.lookup* (*Mapping.filter P m*) *k* ≠ *None* ⟹
Mapping.lookup (*Mapping.filter P m*) *k* = *Mapping.lookup m k*
⟨*proof*⟩

lemma *fold_Mapping_filter_Some_None*: *Mapping.lookup ts as* = *Some t* ⟹
as ∈ *X* ⟹ (*t*, *X*) ∈ *set ds* ⟹
Mapping.lookup (*fold* (λ(*t*, *X*) *ts*. *Mapping.filter* (*filter_cond X ts t*) *ts*) *ds ts*) *as* = *None*
⟨*proof*⟩

lemma *fold_Mapping_filter_Some_Some*: *Mapping.lookup ts as* = *Some t* ⟹
$(\bigwedge X.\ (t,\ X) \in set\ ds \Longrightarrow as \notin X) \Longrightarrow$
Mapping.lookup (*fold* (λ(*t*, *X*) *ts*. *Mapping.filter* (*filter_cond X ts t*) *ts*) *ds ts*) *as* = *Some t*
⟨*proof*⟩

fun *shift_end* :: *args* ⇒ *ts* ⇒ *'a mmsaux* ⇒ *'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* (λ(*t*, *X*). *enat* (*nt* − *t*) > *right I*) *data_prev*;
(*data_in*, *discard*) = *takedropWhile_queue* (λ(*t*, *X*). *enat* (*nt* − *t*) > *right I*) *data_in*;
tuple_in = *fold* (λ(*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* ≥ *cur*
shows *valid_mmsaux args cur* (*shift_end args nt*
(*ot*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))
(*filter* (λ(*t*, *rel*). *enat* (*nt* − *t*) ≤ *right* (*args_ivl args*)) *auxlist*)
⟨*proof*⟩

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*)
⟨*proof*⟩

setup_lifting *type_definition_mapping*

lift_definition *upd_set* :: (*'a*, *'b*) *mapping* ⇒ (*'a* ⇒ *'b*) ⇒ *'a set* ⇒ (*'a*, *'b*) *mapping* **is**
λ*m f X a*. *if a* ∈ *X then Some* (*f a*) *else m a* ⟨*proof*⟩

lemma *Mapping_lookup_upd_set*: *Mapping.lookup* (*upd_set m f X*) *a* =
(*if a* ∈ *X then Some* (*f a*) *else Mapping.lookup m a*)
⟨*proof*⟩

lemma *Mapping_upd_set_keys*: *Mapping.keys* (*upd_set m f X*) = *Mapping.keys m* ∪ *X*
⟨*proof*⟩

lift_definition *upd_keys_on* :: ('*a*, '*b*) *mapping* ⇒ ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a set* ⇒
('*a*, '*b*) *mapping* **is**
λ*m f X a*. *case Mapping.lookup m a of Some b* ⇒ *Some* (*if a* ∈ *X then f a b else b*)
| *None* ⇒ *None* ⟨*proof*⟩

lemma *Mapping_lookup_upd_keys_on*: *Mapping.lookup* (*upd_keys_on m f X*) *a* =
(*case Mapping.lookup m a of Some b* ⇒ *Some* (*if a* ∈ *X then f a b else b*) | *None* ⇒ *None*)
⟨*proof*⟩

lemma *Mapping_upd_keys_sub*: *Mapping.keys* (*upd_keys_on m f X*) = *Mapping.keys m*
⟨*proof*⟩

lemma *fold_append_queue_rep*: *linearize* (*fold* (λ*x q*. *append_queue x q*) *xs q*) = *linearize q* @ *xs*
⟨*proof*⟩

lemma *Max_Un_absorb*:
assumes *finite X X* ≠ {} *finite Y* (⋀*x y*. *y* ∈ *Y* ⟹ *x* ∈ *X* ⟹ *y* ≤ *x*)
shows *Max* (*X* ∪ *Y*) = *Max X*
⟨*proof*⟩

lemma *Mapping_lookup_fold_upd_set_idle*: {(*t*, *X*) ∈ *set xs*. *as* ∈ *Z X t*} = {} ⟹
Mapping.lookup (*fold* (λ(*t*, *X*) *m*. *upd_set m* (λ_. *t*) (*Z X t*)) *xs m*) *as* = *Mapping.lookup m as*
⟨*proof*⟩

lemma *Mapping_lookup_fold_upd_set_max*: {(*t*, *X*) ∈ *set xs*. *as* ∈ *Z X t*} ≠ {} ⟹
sorted (*map fst xs*) ⟹
Mapping.lookup (*fold* (λ(*t*, *X*) *m*. *upd_set m* (λ_. *t*) (*Z X t*)) *xs m*) *as* =
Some (*Max* (*fst* ‘ {(*t*, *X*) ∈ *set xs*. *as* ∈ *Z X t*}))
⟨*proof*⟩

fun *add_new_ts_mmsaux'* :: *args* ⇒ *ts* ⇒ '*a mmsaux* ⇒ '*a mmsaux* **where**
add_new_ts_mmsaux' args nt (*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
(*let I* = *args_ivl args*;
(*data_prev*, *move*) = *takedropWhile_queue* (λ(*t*, *X*). *nt* − *t* ≥ *left I*) *data_prev*;
data_in = *fold* (λ(*t*, *X*) *data_in*. *append_queue* (*t*, *X*) *data_in*) *move data_in*;
tuple_in = *fold* (λ(*t*, *X*) *tuple_in*. *upd_set tuple_in* (λ_. *t*)
{*as* ∈ *X*. *valid_tuple tuple_since* (*t*, *as*)}) *move tuple_in in*
(*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))

lemma *Mapping_keys_fold_upd_set*: *k* ∈ *Mapping.keys* (*fold* (λ(*t*, *X*) *m*. *upd_set m* (λ_. *t*) (*Z t X*))
xs m) ⟹ *k* ∈ *Mapping.keys m* ∨ (∃ (*t*, *X*) ∈ *set xs*. *k* ∈ *Z t X*)
⟨*proof*⟩

lemma *valid_add_new_ts_mmsaux'_unfolded*:
assumes *valid_before*: *valid_mmsaux args cur*
(*ot*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
and *nt_mono*: *nt* ≥ *cur*
shows *valid_mmsaux args nt* (*add_new_ts_mmsaux' args nt*
(*ot*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*)) *auxlist*
⟨*proof*⟩

lemma *valid_add_new_ts_mmsaux'*: *valid_mmsaux args cur aux auxlist* ⟹ *nt* ≥ *cur* ⟹
valid_mmsaux args nt (*add_new_ts_mmsaux' args nt aux*) *auxlist*
⟨*proof*⟩

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*)
⟨*proof*⟩

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* (λ*as* _. *proj_tuple_in_join pos maskL as X*) *tuple_in*;
tuple_since = *Mapping.filter* (λ*as* _. *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* (λ*as* _. *proj_tuple_in_join pos maskL as X*) *tuple_in*;
tuple_since = *Mapping.filter* (λ*as* _. *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*: (⋀*k v*. *k* ∈ *Mapping.keys m* ⟹ *f k v* = *f' k v*)
shows *Mapping.filter f m* = *Mapping.filter f' m*
⟨*proof*⟩

lemma *join_mmsaux_abs_eq*:
assumes *valid_before*: *valid_mmsaux args cur*
(*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
and *table_left*: *table* (*args_n args*) (*args_L args*) *X*
shows *join_mmsaux args X* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
join_mmsaux_abs args X (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*)
⟨*proof*⟩

lemma *valid_join_mmsaux_unfolded*:
assumes *valid_before*: *valid_mmsaux args cur*
(*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
and *table_left'*: *table* (*args_n args*) (*args_L args*) *X*
shows *valid_mmsaux args cur*
(*join_mmsaux args X* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*))
(*map* (λ(*t*, *rel*). (*t*, *join rel* (*args_pos args*) *X*)) *auxlist*)
⟨*proof*⟩

lemma *valid_join_mmsaux*: *valid_mmsaux args cur aux auxlist* ⟹
table (*args_n args*) (*args_L args*) *X* ⟹ *valid_mmsaux args cur*
(*join_mmsaux args X aux*) (*map* (λ(*t*, *rel*). (*t*, *join rel* (*args_pos args*) *X*)) *auxlist*)
⟨*proof*⟩

fun *gc_mmsaux* :: *'a mmsaux* ⇒ *'a mmsaux* **where**
gc_mmsaux (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
(*let all_tuples* = ⋃(*snd* ‘ (*set* (*linearize data_prev*) ∪ *set* (*linearize data_in*)));
tuple_since' = *Mapping.filter* (λ*as* _. *as* ∈ *all_tuples*) *tuple_since in*
(*nt*, *nt*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since'*))

lemma *valid_gc_mmsaux_unfolded*:
assumes *valid_before*: *valid_mmsaux args cur* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*,
tuple_in, *tuple_since*) *ys*
shows *valid_mmsaux args cur* (*gc_mmsaux* (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*,
tuple_in, *tuple_since*)) *ys*
⟨*proof*⟩

lemma *valid_gc_mmsaux*: *valid_mmsaux args cur aux ys* ⟹ *valid_mmsaux args cur* (*gc_mmsaux aux*)
ys
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *valid_gc_join_mmsaux*:
assumes *valid_mmsaux args cur aux auxlist table* (*args_n args*) (*args_L args*) *rel1*
shows *valid_mmsaux args cur* (*gc_join_mmsaux args rel1 aux*)
(*map* (λ(*t*, *rel*). (*t*, *join rel* (*args_pos args*) *rel1*)) *auxlist*)
⟨*proof*⟩

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*⟩

lemma *valid_add_new_table_mmsaux*:
assumes *valid_before*: *valid_mmsaux args cur aux auxlist*
and *table_X*: *table* (*args_n args*) (*args_R args*) *X*
shows *valid_mmsaux args cur* (*add_new_table_mmsaux args X aux*)
(*case auxlist of*

[] => [(cur, X)]
| ((t, y) # ts) ⇒ if t = cur then (t, y ∪ X) # ts else (cur, X) # auxlist)
⟨proof⟩

lemma *foldr_ts_tuple_rel*:
as ∈ *foldr* (∪) (*concat* (*map* (λ(*t*, *rel*). *if P t then* [*rel*] *else* []) *auxlist*)) {} ⟷
(∃ *t*. (*t*, *as*) ∈ *ts_tuple_rel* (*set auxlist*) ∧ *P t*)
⟨proof⟩

lemma *image_snd*: (*a*, *b*) ∈ *X* ⟹ *b* ∈ *snd* ‘ *X*
⟨proof⟩

fun *result_mmsaux* :: *args* ⇒ 'a *mmsaux* ⇒ 'a *table* **where**
result_mmsaux args (*nt*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
Mapping.keys tuple_in

lemma *valid_result_mmsaux_unfolded*:
assumes *valid_mmsaux args cur*
(*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) *auxlist*
shows *result_mmsaux args* (*t*, *gc*, *maskL*, *maskR*, *data_prev*, *data_in*, *tuple_in*, *tuple_since*) =
foldr (∪) [*rel*. (*t*, *rel*) ← *auxlist*, *left* (*args_ivl args*) ≤ *cur* − *t*] {}
⟨proof⟩

lemma *valid_result_mmsaux*: *valid_mmsaux args cur aux auxlist* ⟹
result_mmsaux args aux = *foldr* (∪) [*rel*. (*t*, *rel*) ← *auxlist*, *left* (*args_ivl args*) ≤ *cur* − *t*] {}
⟨proof⟩

interpretation *default_msaux*: *msaux valid_mmsaux init_mmsaux add_new_ts_mmsaux gc_join_mmsaux*
add_new_table_mmsaux result_mmsaux
⟨proof⟩

7.3 Optimized data structure for Until

type_synonym *tp* = *nat*

type_synonym 'a *mmuaux* = *tp* × *ts queue* × *nat* × *bool list* × *bool list* ×
('a *tuple*, *tp*) *mapping* × (*tp*, ('a *tuple*, *ts* + *tp*) *mapping*) *mapping* × 'a *table list* × *nat*

definition *tstp_lt* :: *ts* + *tp* ⇒ *ts* ⇒ *tp* ⇒ *bool* **where**
tstp_lt tstp ts tp = *case_sum* (λ*ts'*. *ts'* ≤ *ts*) (λ*tp'*. *tp'* < *tp*) *tstp*

definition *tstp_le* :: *ts* + *tp* ⇒ *ts* ⇒ *tp* ⇒ *bool* **where**
tstp_le tstp ts tp = *case_sum* (λ*ts'*. *ts'* ≤ *ts*) (λ*tp'*. *tp'* ≤ *tp*) *tstp*

definition *ts_tp_lt* :: *ts* ⇒ *tp* ⇒ *ts* + *tp* ⇒ *bool* **where**
ts_tp_lt ts tp tstp = *case_sum* (λ*ts'*. *ts* ≤ *ts'*) (λ*tp'*. *tp* < *tp'*) *tstp*

definition *ts_tp_lt'* :: *ts* ⇒ *tp* ⇒ *ts* + *tp* ⇒ *bool* **where**
ts_tp_lt' ts tp tstp = *case_sum* (λ*ts'*. *ts* < *ts'*) (λ*tp'*. *tp* ≤ *tp'*) *tstp*

definition *ts_tp_le* :: *ts* ⇒ *tp* ⇒ *ts* + *tp* ⇒ *bool* **where**
ts_tp_le ts tp tstp = *case_sum* (λ*ts'*. *ts* ≤ *ts'*) (λ*tp'*. *tp* ≤ *tp'*) *tstp*

fun *max_tstp* :: *ts* + *tp* ⇒ *ts* + *tp* ⇒ *ts* + *tp* **where**
max_tstp (*Inl ts*) (*Inl ts'*) = *Inl* (*max ts ts'*)
| *max_tstp* (*Inr tp*) (*Inr tp'*) = *Inr* (*max tp tp'*)
| *max_tstp* (*Inl ts*) _ = *Inl ts*
| *max_tstp* _ (*Inl ts*) = *Inl ts*

lemma *max_tstp_idem*: *max_tstp* (*max_tstp x y*) *y* = *max_tstp x y*
⟨*proof*⟩

lemma *max_tstp_idem′*: *max_tstp x* (*max_tstp x y*) = *max_tstp x y*
⟨*proof*⟩

lemma *max_tstp_d_d*: *max_tstp d d* = *d*
⟨*proof*⟩

lemma *max_cases*: (*max a b* = *a* ⟹ *P*) ⟹ (*max a b* = *b* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *max_tstpE*: *isl tstp* ⟷ *isl tstp′* ⟹ (*max_tstp tstp tstp′* = *tstp* ⟹ *P*) ⟹
(*max_tstp tstp tstp′* = *tstp′* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *max_tstp_intro*: *tstp_lt tstp ts tp* ⟹ *tstp_lt tstp′ ts tp* ⟹ *isl tstp* ⟷ *isl tstp′* ⟹
tstp_lt (*max_tstp tstp tstp′*) *ts tp*
⟨*proof*⟩

lemma *max_tstp_intro′*: *isl tstp* ⟷ *isl tstp′* ⟹
ts_tp_le ts′ tp′ tstp ⟹ *ts_tp_le ts′ tp′* (*max_tstp tstp tstp′*)
⟨*proof*⟩

lemma *max_tstp_intro″*: *isl tstp* ⟷ *isl tstp′* ⟹
ts_tp_le ts′ tp′ tstp′ ⟹ *ts_tp_le ts′ tp′* (*max_tstp tstp tstp′*)
⟨*proof*⟩

lemma *max_tstp_intro‴*: *isl tstp* ⟷ *isl tstp′* ⟹
ts_tp_lt′ ts′ tp′ tstp ⟹ *ts_tp_lt′ ts′ tp′* (*max_tstp tstp tstp′*)
⟨*proof*⟩

lemma *max_tstp_intro⁗*: *isl tstp* ⟷ *isl tstp′* ⟹
ts_tp_lt′ ts′ tp′ tstp′ ⟹ *ts_tp_lt′ ts′ tp′* (*max_tstp tstp tstp′*)
⟨*proof*⟩

lemma *max_tstp_isl*: *isl tstp* ⟷ *isl tstp′* ⟹ *isl* (*max_tstp tstp tstp′*) ⟷ *isl tstp*
⟨*proof*⟩

definition *filter_a1_map* :: *bool* ⇒ *tp* ⇒ (′*a tuple*, *tp*) *mapping* ⇒ ′*a table* **where**
filter_a1_map pos tp a1_map =
{*xs* ∈ *Mapping.keys a1_map*. *case Mapping.lookup a1_map xs of Some tp′* ⇒
(*pos* ⟶ *tp′* ≤ *tp*) ∧ (¬*pos* ⟶ *tp* ≤ *tp′*)}

definition *filter_a2_map* :: *ℐ* ⇒ *ts* ⇒ *tp* ⇒ (*tp*, (′*a tuple*, *ts* + *tp*) *mapping*) *mapping* ⇒
′*a table* **where**
filter_a2_map I ts tp a2_map = {*xs*. ∃ *tp′* ≤ *tp*. (*case Mapping.lookup a2_map tp′ of Some m* ⇒
(*case Mapping.lookup m xs of Some tstp* ⇒ *ts_tp_lt′ ts tp tstp* | _ ⇒ *False*)
| _ ⇒ *False*)}

fun *triple_eq_pair* :: (′*a* × ′*b* × ′*c*) ⇒ (′*a* × ′*d*) ⇒ (′*d* ⇒ ′*b*) ⇒ (′*a* ⇒ ′*d* ⇒ ′*c*) ⇒ *bool* **where**
triple_eq_pair (*t*, *a1*, *a2*) (*ts′*, *tp′*) *f g* ⟷ *t* = *ts′* ∧ *a1* = *f tp′* ∧ *a2* = *g ts′ tp′*

fun *valid_mmuaux′* :: *args* ⇒ *ts* ⇒ *ts* ⇒ ′*a mmuaux* ⇒ ′*a muaux* ⇒ *bool* **where**
valid_mmuaux′ args cur dt (*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*,
done, *done_length*) *auxlist* ⟷
args_L args ⊆ *args_R args* ∧

```
    maskL = join_mask (args_n args) (args_L args) ∧
    maskR = join_mask (args_n args) (args_R args) ∧
    len ≤ tp ∧
    length (linearize tss) = len ∧ sorted (linearize tss) ∧
    (∀ t ∈ set (linearize tss). t ≤ cur ∧ enat cur ≤ enat t + right (args_ivl args)) ∧
    table (args_n args) (args_L args) (Mapping.keys a1_map) ∧
    Mapping.keys a2_map = {tp − len..tp} ∧
    (∀ xs ∈ Mapping.keys a1_map. case Mapping.lookup a1_map xs of Some tp' ⇒ tp' < tp) ∧
    (∀ tp' ∈ Mapping.keys a2_map. case Mapping.lookup a2_map tp' of Some m ⇒
      table (args_n args) (args_R args) (Mapping.keys m) ∧
      (∀ xs ∈ Mapping.keys m. case Mapping.lookup m xs of Some tstp ⇒
      tstp_lt tstp (cur − (left (args_ivl args) − 1)) tp ∧ (isl tstp ⟷ left (args_ivl args) > 0))) ∧
    length done = done_length ∧ length done + len = length auxlist ∧
    rev done = map proj_thd (take (length done) auxlist) ∧
    (∀ x ∈ set (take (length done) auxlist). check_before (args_ivl args) dt x) ∧
    sorted (map fst auxlist) ∧
    list_all2 (λx y. triple_eq_pair x y (λtp'. filter_a1_map (args_pos args) tp' a1_map)
      (λts' tp'. filter_a2_map (args_ivl args) ts' tp' a2_map)) (drop (length done) auxlist)
      (zip (linearize tss) [tp − len..<tp])

definition valid_mmuaux :: args ⇒ ts ⇒ 'a mmuaux ⇒ 'a muaux ⇒
  bool where
  valid_mmuaux args cur = valid_mmuaux' args cur cur

fun eval_step_mmuaux :: 'a mmuaux ⇒ 'a mmuaux where
  eval_step_mmuaux (tp, tss, len, maskL, maskR, a1_map, a2_map,
    done, done_length) = (case safe_hd tss of (Some ts, tss') ⇒
      (case Mapping.lookup a2_map (tp − len) of Some m ⇒
        let m = Mapping.filter (λ_ tstp. ts_tp_lt' ts (tp − len) tstp) m;
        T = Mapping.keys m;
        a2_map = Mapping.update (tp − len + 1)
          (case Mapping.lookup a2_map (tp − len + 1) of Some m' ⇒
          Mapping.combine (λtstp tstp'. max_tstp tstp tstp') m m') a2_map;
        a2_map = Mapping.delete (tp − len) a2_map in
        (tp, tl_queue tss', len − 1, maskL, maskR, a1_map, a2_map,
        T # done, done_length + 1)))

lemma Mapping_update_keys: Mapping.keys (Mapping.update a b m) = Mapping.keys m ∪ {a}
  ⟨proof⟩

lemma drop_is_Cons_take: drop n xs = y # ys ⟹ take (Suc n) xs = take n xs @ [y]
⟨proof⟩

lemma list_all2_weaken: list_all2 f xs ys ⟹
  (⋀x y. (x, y) ∈ set (zip xs ys) ⟹ f x y ⟹ f' x y) ⟹ list_all2 f' xs ys
  ⟨proof⟩

lemma Mapping_lookup_delete: Mapping.lookup (Mapping.delete k m) k' =
  (if k = k' then None else Mapping.lookup m k')
  ⟨proof⟩

lemma Mapping_lookup_update: Mapping.lookup (Mapping.update k v m) k' =
  (if k = k' then Some v else Mapping.lookup m k')
  ⟨proof⟩

lemma hd_le_set: sorted xs ⟹ xs ≠ [] ⟹ x ∈ set xs ⟹ hd xs ≤ x
  ⟨proof⟩
```

lemma *Mapping_lookup_combineE*: *Mapping.lookup* (*Mapping.combine f m m'*) *k* = *Some v* ⟹
(*Mapping.lookup m k* = *Some v* ⟹ *P*) ⟹
(*Mapping.lookup m' k* = *Some v* ⟹ *P*) ⟹
(⋀*v' v''*. *Mapping.lookup m k* = *Some v'* ⟹ *Mapping.lookup m' k* = *Some v''* ⟹
f v' v'' = *v* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *Mapping_keys_filterI*: *Mapping.lookup m k* = *Some v* ⟹ *f k v* ⟹
k ∈ *Mapping.keys* (*Mapping.filter f m*)
⟨*proof*⟩

lemma *Mapping_keys_filterD*: *k* ∈ *Mapping.keys* (*Mapping.filter f m*) ⟹
∃ *v*. *Mapping.lookup m k* = *Some v* ∧ *f k v*
⟨*proof*⟩

fun *lin_ts_mmuaux* :: *'a mmuaux* ⇒ *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* ∧
lin_ts_mmuaux (*eval_step_mmuaux aux*) = *tss''*
⟨*proof*⟩

lemma *done_empty_valid_mmuaux'_intro*:
assumes *valid_mmuaux' args cur dt*
(*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*, *done*, *done_length*) *auxlist*
shows *valid_mmuaux' args cur dt'*
(*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*, [], *0*)
(*drop* (*length done*) *auxlist*)
⟨*proof*⟩

lemma *valid_mmuaux'_mono*:
assumes *valid_mmuaux' args cur dt aux auxlist dt* ≤ *dt'*
shows *valid_mmuaux' args cur dt' aux auxlist*
⟨*proof*⟩

lemma *valid_foldl_eval_step_mmuaux'*:
assumes *valid_before*: *valid_mmuaux' args cur dt aux auxlist*
lin_ts_mmuaux aux = *tss* @ *tss'*
⋀*ts*. *ts* ∈ *set* (*take* (*length tss*) (*lin_ts_mmuaux aux*)) ⟹ *enat ts* + *right* (*args_ivl args*) < *dt*
shows *valid_mmuaux' args cur dt* (*foldl* (λ*aux* _. *eval_step_mmuaux aux*) *aux tss*) *auxlist* ∧
lin_ts_mmuaux (*foldl* (λ*aux* _. *eval_step_mmuaux aux*) *aux tss*) = *tss'*
⟨*proof*⟩

lemma *sorted_dropWhile_filter*: *sorted xs* ⟹ *dropWhile* (λ*t*. *enat t* + *right I* < *enat nt*) *xs* =
filter (λ*t*. ¬*enat t* + *right I* < *enat nt*) *xs*
⟨*proof*⟩

fun *shift_mmuaux* :: *args* ⇒ *ts* ⇒ *'a mmuaux* ⇒ *'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* (λ*t*. *enat t* + *right* (*args_ivl args*) < *enat nt*) *tss*) *in*
foldl (λ*aux* _. *eval_step_mmuaux aux*) (*tp*, *tss*, *len*, *maskL*, *maskR*,
a1_map, *a2_map*, *done*, *done_length*) *tss_list*)

lemma *valid_shift_mmuaux'*:

assumes *valid_mmuaux' args cur cur aux auxlist nt* $\geq$ *cur*
shows *valid_mmuaux' args cur nt (shift_mmuaux args nt aux) auxlist* $\wedge$
(∀ *ts* ∈ *set* (*lin_ts_mmuaux* (*shift_mmuaux args nt aux*)). ¬*enat ts* + *right* (*args_ivl args*) < *nt*)
⟨*proof*⟩

lift_definition *upd_set'* :: ('*a*, '*b*) *mapping* ⇒ '*b* ⇒ ('*b* ⇒ '*b*) ⇒ '*a set* ⇒ ('*a*, '*b*) *mapping* **is**
λ*m d f X a*. (*if a* ∈ *X then* (*case Mapping.lookup m a of Some b* ⇒ *Some* (*f b*) | *None* ⇒ *Some d*)
else Mapping.lookup m a) ⟨*proof*⟩

lemma *upd_set'_lookup*: *Mapping.lookup* (*upd_set' m d f X*) *a* = (*if a* ∈ *X then*
(*case Mapping.lookup m a of Some b* ⇒ *Some* (*f b*) | *None* ⇒ *Some d*) *else Mapping.lookup m a*)
⟨*proof*⟩

lemma *upd_set'_keys*: *Mapping.keys* (*upd_set' m d f X*) = *Mapping.keys m* ∪ *X*
⟨*proof*⟩

lift_definition *upd_nested* :: ('*a*, ('*b*, '*c*) *mapping*) *mapping* ⇒
'*c* ⇒ ('*c* ⇒ '*c*) ⇒ ('*a* × '*b*) *set* ⇒ ('*a*, ('*b*, '*c*) *mapping*) *mapping* **is**
λ*m d f X a. case Mapping.lookup m a of Some m'* ⇒ *Some* (*upd_set' m' d f* {*b*. (*a*, *b*) ∈ *X*})
| *None* ⇒ *if a* ∈ *fst* ‘ *X then Some* (*upd_set' Mapping.empty d f* {*b*. (*a*, *b*) ∈ *X*}) *else None* ⟨*proof*⟩

lemma *upd_nested_lookup*: *Mapping.lookup* (*upd_nested m d f X*) *a* =
(*case Mapping.lookup m a of Some m'* ⇒ *Some* (*upd_set' m' d f* {*b*. (*a*, *b*) ∈ *X*})
| *None* ⇒ *if a* ∈ *fst* ‘ *X then Some* (*upd_set' Mapping.empty d f* {*b*. (*a*, *b*) ∈ *X*}) *else None*)
⟨*proof*⟩

lemma *upd_nested_keys*: *Mapping.keys* (*upd_nested m d f X*) = *Mapping.keys m* ∪ *fst* ‘ *X*
⟨*proof*⟩

fun *add_new_mmuaux* :: *args* ⇒ '*a table* ⇒ '*a table* ⇒ *ts* ⇒ '*a mmuaux* ⇒ '*a mmuaux* **where**
add_new_mmuaux args rel1 rel2 nt aux =
(*let* (*tp*, *tss*, *len*, *maskL*, *maskR*, *a1_map*, *a2_map*, *done*, *done_length*) =
shift_mmuaux args nt aux;
I = *args_ivl args*; *pos* = *args_pos args*;
new_tstp = (*if left I* = *0 then Inr tp else Inl* (*nt* − (*left I* − *1*)));
tmp = ⋃((λ*as*. *case Mapping.lookup a1_map* (*proj_tuple maskL as*) *of None* ⇒
(*if* ¬*pos then* {(*tp* − *len*, *as*)} *else* {})
| *Some tp'* ⇒ *if pos then* {(*max* (*tp* − *len*) *tp'*, *as*)}
else {(*max* (*tp* − *len*) (*tp'* + *1*), *as*)}) ‘ *rel2*) ∪ (*if left I* = *0 then* {*tp*} × *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* (λ*as* _. *as* ∈ *rel1*)
(*upd_set a1_map* (λ_. *tp*) (*rel1* − *Mapping.keys a1_map*)) *else upd_set a1_map* (λ_. *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*: (λ*x*. *fst* (*case x of* (*t*, *a1*, *a2*) ⇒ (*t*, *y t a1 a2*, *z t a1 a2*))) = *fst*
⟨*proof*⟩

lemma *list_all2_in_setE*: *list_all2 P xs ys* ⟹ *x* ∈ *set xs* ⟹ (⋀*y*. *y* ∈ *set ys* ⟹ *P x y* ⟹ *Q*) ⟹ *Q*
⟨*proof*⟩

lemma *list_all2_zip*: *list_all2* (λ*x y*. *triple_eq_pair x y f g*) *xs* (*zip ys zs*) ⟹
(⋀*y*. *y* ∈ *set ys* ⟹ *Q y*) ⟹ *x* ∈ *set xs* ⟹ *Q* (*fst x*)
⟨*proof*⟩

lemma *list_appendE*: *xs* = *ys* @ *zs* ⟹ *x* ∈ *set xs* ⟹
(*x* ∈ *set ys* ⟹ *P*) ⟹ (*x* ∈ *set zs* ⟹ *P*) ⟹ *P*

⟨*proof*⟩

lemma *take_takeWhile*: $n \leq length\ ys \Longrightarrow$
$(\bigwedge y.\ y \in set\ (take\ n\ ys) \Longrightarrow P\ y) \Longrightarrow$
$(\bigwedge y.\ y \in set\ (drop\ n\ ys) \Longrightarrow \neg P\ y) \Longrightarrow$
$take\ n\ ys = takeWhile\ P\ ys$
⟨*proof*⟩

lemma *valid_add_new_mmuaux*:
assumes *valid_before*: *valid_mmuaux args cur aux auxlist*
and *tabs*: *table* (*args_n args*) (*args_L args*) *rel1 table* (*args_n args*) (*args_R args*) *rel2*
and *nt_mono*: $nt \geq cur$
shows *valid_mmuaux args nt* (*add_new_mmuaux args rel1 rel2 nt aux*)
(*update_until args rel1 rel2 nt auxlist*)
⟨*proof*⟩

lemma *list_all2_check_before*: *list_all2* $(\lambda x\ y.$ *triple_eq_pair x y f g*) *xs* (*zip ys zs*) $\Longrightarrow$
$(\bigwedge y.\ y \in set\ ys \Longrightarrow \neg enat\ y + right\ I < nt) \Longrightarrow x \in set\ xs \Longrightarrow \neg check\_before\ I\ nt\ x$
⟨*proof*⟩

fun *eval_mmuaux* :: *args* ⇒ *ts* ⇒ *'a mmuaux* ⇒ *'a table list* × *'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'* ∧ *valid_mmuaux args cur aux' auxlist'*
⟨*proof*⟩

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*: $L \subseteq R \Longrightarrow$ *valid_mmuaux* (*init_args I n L R b*) *0*
(*init_mmuaux* (*init_args I n L R b*)) []
⟨*proof*⟩

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*
⟨*proof*⟩

8 Instantiation of the generic algorithm and code setup

lemma [*code_unfold del*, *symmetric*, *code_post del*]: *card* ≡ *Cardinality.card'* ⟨*proof*⟩
declare [[*code drop*: *card*]] *Set_Impl.card_code*[*code*]

instantiation *enat* :: *set_impl* **begin**
definition *set_impl_enat* :: (*enat*, *set_impl*) *phantom* **where**
set_impl_enat = *phantom set_RBT*

```
instance ⟨proof⟩
end

derive ccompare Formula.trm
derive (eq) ceq Formula.trm
derive (rbt) set_impl Formula.trm
derive (eq) ceq Monitor.mregex
derive ccompare Monitor.mregex
derive (rbt) set_impl Monitor.mregex
derive (rbt) mapping_impl Monitor.mregex
derive (no) cenum Monitor.mregex
derive (rbt) set_impl event_data
derive (rbt) mapping_impl event_data

definition add_new_mmuaux' :: args ⇒ event_data table ⇒ event_data table ⇒ ts ⇒ event_data mmuaux ⇒
  event_data mmuaux where
  add_new_mmuaux' = add_new_mmuaux

interpretation muaux valid_mmuaux init_mmuaux add_new_mmuaux' length_mmuaux eval_mmuaux
  ⟨proof⟩

type_synonym 'a vmsaux = nat × (nat × 'a table) list

definition valid_vmsaux :: args ⇒ nat ⇒ event_data vmsaux ⇒
  (nat × event_data table) list ⇒ bool where
  valid_vmsaux = (λ_ cur (t, aux) auxlist. t = cur ∧ aux = auxlist)

definition init_vmsaux :: args ⇒ event_data vmsaux where
  init_vmsaux = (λ_. (0, []))

definition add_new_ts_vmsaux :: args ⇒ nat ⇒ event_data vmsaux ⇒ event_data vmsaux where
  add_new_ts_vmsaux = (λargs nt (t, auxlist). (nt, filter (λ(t, rel).
    enat (nt − t) ≤ right (args_ivl args)) auxlist))

definition join_vmsaux :: args ⇒ event_data table ⇒ event_data vmsaux ⇒ event_data vmsaux where
  join_vmsaux = (λargs rel1 (t, auxlist). (t, map (λ(t, rel).
    (t, join rel (args_pos args) rel1)) auxlist))

definition add_new_table_vmsaux :: args ⇒ event_data table ⇒ event_data vmsaux ⇒
  event_data vmsaux where
  add_new_table_vmsaux = (λargs rel2 (cur, auxlist). (cur, (case auxlist of
    [] => [(cur, rel2)]
  | ((t, y) # ts) ⇒ if t = cur then (t, y ∪ rel2) # ts else (cur, rel2) # auxlist)))

definition result_vmsaux :: args ⇒ event_data vmsaux ⇒ event_data table where
  result_vmsaux = (λargs (cur, auxlist).
    foldr (∪) [rel. (t, rel) ← auxlist, left (args_ivl args) ≤ cur − t] {})

type_synonym 'a vmuaux = nat × (nat × 'a table × 'a table) list

definition valid_vmuaux :: args ⇒ nat ⇒ event_data vmuaux ⇒
  (nat × event_data table × event_data table) list ⇒ bool where
  valid_vmuaux = (λ_ cur (t, aux) auxlist. t = cur ∧ aux = auxlist)

definition init_vmuaux :: args ⇒ event_data vmuaux where
  init_vmuaux = (λ_. (0, []))
```

definition *add_new_vmuaux* :: *args* ⇒ *event_data table* ⇒ *event_data table* ⇒ *nat* ⇒
event_data vmuaux ⇒ *event_data vmuaux* **where**
add_new_vmuaux = (λ*args rel1 rel2 nt* (*t*, *auxlist*). (*nt*, *update_until args rel1 rel2 nt auxlist*))

definition *length_vmuaux* :: *args* ⇒ *event_data vmuaux* ⇒ *nat* **where**
length_vmuaux = (λ_ (_, *auxlist*). *length auxlist*)

definition *eval_vmuaux* :: *args* ⇒ *nat* ⇒ *event_data vmuaux* ⇒
event_data table list × *event_data vmuaux* **where**
eval_vmuaux = (λ*args nt* (*t*, *auxlist*).
(*let* (*res*, *auxlist*′) = *eval_until* (*args_ivl args*) *nt auxlist in* (*res*, (*t*, *auxlist*′))))

global_interpretation *verimon_maux*: *maux valid_vmsaux init_vmsaux add_new_ts_vmsaux join_vmsaux*
add_new_table_vmsaux result_vmsaux valid_vmuaux init_vmuaux add_new_vmuaux length_vmuaux
eval_vmuaux
defines *vminit0* = *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* ⇒ _)
⟨*proof*⟩

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* ⇒ _)
⟨*proof*⟩

lemma *image_these*: *f* ‘ *Option.these X* = *Option.these* (*map_option f* ‘ *X*)
⟨*proof*⟩

thm *default_maux.meval.simps*(*2*)

lemma *meval_MPred*: *meval n t db* (*MPred e ts*) =
(*case Mapping.lookup db e of None* ⇒ [{}] | *Some Xs* ⇒ *map* (λ*X*. ⋃ *v* ∈ *X*.
(*set_option* (*map_option* (λ*f*. *Table.tabulate f 0 n*) (*match ts v*)))) *Xs*, *MPred e ts*)
⟨*proof*⟩

lemmas *meval_code*[*code*] = *default_maux.meval.simps*(*1*) *meval_MPred default_maux.meval.simps*(*3*−)

definition *mk_db* :: (*Formula.name* × *event_data list set*) *list* ⇒ _ **where**
mk_db t = *Monitor.mk_db* (⋃ *n* ∈ *set* (*map fst t*). (λ*v*. (*n*, *v*)) ‘ *the* (*map_of t n*))

definition *rbt_fold* :: _ ⇒ *event_data tuple set_rbt* ⇒ _ ⇒ _ **where**
rbt_fold = *RBT_Set2.fold*

definition *rbt_empty* :: *event_data list set_rbt* **where**
rbt_empty = *RBT_Set2.empty*

definition *rbt_insert* :: _ ⇒ _ ⇒ *event_data list set_rbt* **where**
rbt_insert = *RBT_Set2.insert*

lemma *saturate_commute*:
assumes ⋀*s*. *r* ∈ *g s* ⋀*s*. *g* (*insert r s*) = *g s* ⋀*s*. *r* ∈ *s* ⟹ *h s* = *g s*
and *terminates*: *mono g* ⋀*X*. *X* ⊆ *C* ⟹ *g X* ⊆ *C finite C*
shows *saturate g* {} = *saturate h* {*r*}
⟨*proof*⟩

definition *RPDs_aux* = *saturate* (λ*S*. *S* ∪ ⋃ (*RPD* ‘ *S*))

lemma *RPDs_aux_code*[*code*]:
RPDs_aux S = (*let S'* = *S* ∪ *Set.bind S RPD in if S'* ⊆ *S then S else RPDs_aux S'*)
⟨*proof*⟩

declare *RPDs_code*[*code del*]
lemma *RPDs_code*[*code*]: *RPDs r* = *RPDs_aux* {*r*}
⟨*proof*⟩

definition *LPDs_aux* = *saturate* (λ*S*. *S* ∪ ⋃ (*LPD* ‘ *S*))

lemma *LPDs_aux_code*[*code*]:
LPDs_aux S = (*let S'* = *S* ∪ *Set.bind S LPD in if S'* ⊆ *S then S else LPDs_aux S'*)
⟨*proof*⟩

declare *LPDs_code*[*code del*]

lemma *LPDs_code*[*code*]: *LPDs r* = *LPDs_aux* {*r*}
⟨*proof*⟩

lemma *is_empty_table_unfold* [*code_unfold*]:
X = *empty_table* ⟷ *Set.is_empty X*
empty_table = *X* ⟷ *Set.is_empty X*
Cardinality.eq_set X empty_table ⟷ *Set.is_empty X*
Cardinality.eq_set empty_table X ⟷ *Set.is_empty X*
set_eq X empty_table ⟷ *Set.is_empty X*
set_eq empty_table X ⟷ *Set.is_empty X*
X = (*set_empty impl*) ⟷ *Set.is_empty X*
(*set_empty impl*) = *X* ⟷ *Set.is_empty X*
Cardinality.eq_set X (*set_empty impl*) ⟷ *Set.is_empty X*
Cardinality.eq_set (*set_empty impl*) *X* ⟷ *Set.is_empty X*
set_eq X (*set_empty impl*) ⟷ *Set.is_empty X*
set_eq (*set_empty impl*) *X* ⟷ *Set.is_empty X*
⟨*proof*⟩

lemma *tabulate_rbt_code*[*code*]: *Monitor.mrtabulate* (*xs* :: *mregex list*) *f* =
(*case ID CCOMPARE*(*mregex*) *of None* ⇒ *Code.abort* (*STR* ''*tabulate RBT_Mapping*: *ccompare* = *None*'') (λ_. *Monitor.mrtabulate* (*xs* :: *mregex list*) *f*)
| _ ⇒ *RBT_Mapping* (*RBT_Mapping2.bulkload* (*List.map_filter* (λ*k*. *let fk* = *f k in if fk* = *empty_table then None else Some* (*k*, *fk*)) *xs*)))
⟨*proof*⟩

lemma *combine_Mapping*[*code*]:
fixes *t* :: ('*a* :: *ccompare*, '*b*) *mapping_rbt* **shows**
Mapping.combine f (*RBT_Mapping t*) (*RBT_Mapping u*) =
(*case ID CCOMPARE*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*combine RBT_Mapping*: *ccompare* = *None*'') (λ_. *Mapping.combine f* (*RBT_Mapping t*) (*RBT_Mapping u*))
| *Some* _ ⇒ *RBT_Mapping* (*RBT_Mapping2.join* (λ_. *f*) *t u*))
⟨*proof*⟩

lemma *upd_set_empty*[*simp*]: *upd_set m f* {} = *m*
⟨*proof*⟩

lemma *upd_set_insert*[*simp*]: *upd_set m f* (*insert x A*) = *Mapping.update x* (*f x*) (*upd_set m f A*)
⟨*proof*⟩

lemma *upd_set_fold*:
assumes *finite A*
shows *upd_set m f A* = *Finite_Set.fold* (λ*a*. *Mapping.update a* (*f a*)) *m A*
⟨*proof*⟩

lift_definition *upd_cfi* :: ('*a* ⇒ '*b*) ⇒ ('*a*, ('*a*, '*b*) *mapping*) *comp_fun_idem*
is λ*f a m*. *Mapping.update a* (*f a*) *m*
⟨*proof*⟩

lemma *upd_set_code*[*code*]:
upd_set m f A = (*if finite A then set_fold_cfi* (*upd_cfi f*) *m A else Code.abort* (*STR* ''*upd_set*: *infinite*'') (λ_. *upd_set m f A*))
⟨*proof*⟩

lemma *lexordp_eq_code*[*code*]: *lexordp_eq xs ys* ⟷ (*case xs of* [] ⇒ *True*
| *x* # *xs* ⇒ (*case ys of* [] ⇒ *False*
| *y* # *ys* ⇒ *if x* < *y then True else if x* > *y then False else lexordp_eq xs ys*))
⟨*proof*⟩

definition *filter_set m X t = Mapping.filter (filter_cond X m t) m*

declare [[*code drop*: *shift_end*]]
declare *shift_end.simps*[*folded filter_set_def*, *code*]

lemma *upd_set'_empty*[*simp*]: *upd_set' m d f {} = m*
⟨*proof*⟩

lemma *upd_set'_insert*: *d = f d ⟹ (⋀x. f (f x) = f x) ⟹ upd_set' m d f (insert x A) =*
(let m' = (upd_set' m d f A) in case Mapping.lookup m' x of None ⇒ Mapping.update x d m'
| Some v ⇒ Mapping.update x (f v) m')
⟨*proof*⟩

lemma *upd_set'_aux1*: *upd_set' Mapping.empty d f {b. b = k ∨ (a, b) ∈ A} =*
Mapping.update k d (upd_set' Mapping.empty d f {b. (a, b) ∈ A})
⟨*proof*⟩

lemma *upd_set'_aux2*: *Mapping.lookup m k = None ⟹ upd_set' m d f {b. b = k ∨ (a, b) ∈ A} =*
Mapping.update k d (upd_set' m d f {b. (a, b) ∈ A})
⟨*proof*⟩

lemma *upd_set'_aux3*: *Mapping.lookup m k = Some v ⟹ upd_set' m d f {b. b = k ∨ (a, b) ∈ A} =*
Mapping.update k (f v) (upd_set' m d f {b. (a, b) ∈ A})
⟨*proof*⟩

lemma *upd_set'_aux4*: *k ∉ fst ‘ A ⟹ upd_set' Mapping.empty d f {b. (k, b) ∈ A} = Mapping.empty*
⟨*proof*⟩

lemma *upd_nested_empty*[*simp*]: *upd_nested m d f {} = m*
⟨*proof*⟩

definition *upd_nested_step* :: *'c ⇒ ('c ⇒ 'c) ⇒ 'a × 'b ⇒ ('a, ('b, 'c) mapping) mapping ⇒*
('a, ('b, 'c) mapping) mapping **where**
upd_nested_step d f x m = (case x of (k, k') ⇒
(case Mapping.lookup m k of Some m' ⇒
(case Mapping.lookup m' k' of Some v ⇒ Mapping.update k (Mapping.update k' (f v) m') m
| None ⇒ Mapping.update k (Mapping.update k' d m') m)
| None ⇒ Mapping.update k (Mapping.update k' d Mapping.empty) m))

lemma *upd_nested_insert*:
d = f d ⟹ (⋀x. f (f x) = f x) ⟹ upd_nested m d f (insert x A) =
upd_nested_step d f x (upd_nested m d f A)
⟨*proof*⟩

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*⟩

lift_definition *upd_nested_max_tstp_cfi* ::
ts + tp ⇒ ('a × 'b, ('a, ('b, ts + tp) mapping) mapping) comp_fun_idem
is *λd. upd_nested_step d (max_tstp d)*
⟨*proof*⟩

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'') (λ_. upd_nested_max_tstp m d X))
⟨*proof*⟩

declare [[*code drop*: *add_new_mmuaux'*]]
declare *add_new_mmuaux'_def*[*unfolded add_new_mmuaux.simps*, *folded upd_nested_max_tstp_def*, *code*]

lemma *filter_set_empty*[*simp*]: *filter_set m {} t = m*
⟨*proof*⟩

lemma *filter_set_insert*[*simp*]: *filter_set m (insert x A) t = (let m' = filter_set m A t in*
case Mapping.lookup m' x of Some u ⇒ if t = u then Mapping.delete x m' else m' | _ ⇒ m')
⟨*proof*⟩

lemma *filter_set_fold*:
assumes *finite A*
shows *filter_set m A t = Finite_Set.fold (λa m.*
case Mapping.lookup m a of Some u ⇒ if t = u then Mapping.delete a m else m | _ ⇒ m) m A
⟨*proof*⟩

lift_definition *filter_cfi* :: *'b ⇒ ('a, ('a, 'b) mapping) comp_fun_idem*
is *λt a m.*
case Mapping.lookup m a of Some u ⇒ if t = u then Mapping.delete a m else m | _ ⇒ m
⟨*proof*⟩

lemma *filter_set_code*[*code*]:
filter_set m A t = (if finite A then set_fold_cfi (filter_cfi t) m A else Code.abort (STR ''upd_set: infinite'') (λ_. filter_set m A t))
⟨*proof*⟩

lemma *filter_Mapping*[*code*]:
fixes *t* :: (*'a* :: *ccompare*, *'b*) *mapping_rbt* **shows**
Mapping.filter P (RBT_Mapping t) =
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR ''filter RBT_Mapping: ccompare = None'') (λ_. Mapping.filter P (RBT_Mapping t))
| Some _ ⇒ RBT_Mapping (RBT_Mapping2.filter (case_prod P) t))
⟨*proof*⟩

definition *filter_join pos X m = Mapping.filter (join_filter_cond pos X) m*

declare [[*code drop*: *join_mmsaux*]]
declare *join_mmsaux.simps*[*folded filter_join_def*, *code*]

lemma *filter_join_False_empty*: *filter_join False {} m = m*
⟨*proof*⟩

lemma *filter_join_False_insert*: *filter_join False (insert a A) m =*
filter_join False A (Mapping.delete a m)
⟨*proof*⟩

lemma *filter_join_False*:
assumes *finite A*
shows *filter_join False A m = Finite_Set.fold Mapping.delete m A*
⟨*proof*⟩

lift_definition *filter_not_in_cfi* :: (*'a*, (*'a*, *'b*) *mapping*) *comp_fun_idem* **is** *Mapping.delete*
⟨*proof*⟩

lemma *filter_join_code*[*code*]:
filter_join pos A m =
(*if* ¬*pos* ∧ *finite A* ∧ *card A* < *Mapping.size m then set_fold_cfi filter_not_in_cfi m A*
else Mapping.filter (*join_filter_cond pos A*) *m*)
⟨*proof*⟩

definition *set_minus* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
set_minus X Y = *X* − *Y*

lift_definition *remove_cfi* :: (*'a*, *'a set*) *comp_fun_idem*
is λ*b a*. *a* − {*b*}
⟨*proof*⟩

lemma *set_minus_finite*:
assumes *fin*: *finite Y*
shows *set_minus X Y* = *Finite_Set.fold* (λ*a X*. *X* − {*a*}) *X Y*
⟨*proof*⟩

lemma *set_minus_code*[*code*]: *set_minus X Y* =
(*if finite Y* ∧ *card Y* < *card X then set_fold_cfi remove_cfi X Y else X* − *Y*)
⟨*proof*⟩

declare [[*code drop*: *bin_join*]]
declare *bin_join.simps*[*folded set_minus_def*, *code*]

definition *remove_Union* **where**
remove_Union A X B = *A* − (⋃ *x* ∈ *X*. *B x*)

lemma *remove_Union_finite*:
assumes *finite X*
shows *remove_Union A X B* = *Finite_Set.fold* (λ*x A*. *A* − *B x*) *A X*
⟨*proof*⟩

lift_definition *remove_Union_cfi* :: (*'a* ⇒ *'b set*) ⇒ (*'a*, *'b set*) *comp_fun_idem* **is** λ*B x A*. *A* − *B x*
⟨*proof*⟩

lemma *remove_Union_code*[*code*]: *remove_Union A X B* =
(*if finite X then set_fold_cfi* (*remove_Union_cfi B*) *A X else A* − (⋃ *x* ∈ *X*. *B x*))
⟨*proof*⟩

lemma *tabulate_remdups*: *Mapping.tabulate xs f* = *Mapping.tabulate* (*remdups xs*) *f*
⟨*proof*⟩

lift_definition *clearjunk* :: (*String.literal* × *event_data list set*) *list* ⇒ (*String.literal*, *event_data list set list*) *alist* **is**
λ*t*. *List.map_filter* (λ(*p*, *X*). *if X* = {} *then None else Some* (*p*, [*X*])) (*AList.clearjunk t*)
⟨*proof*⟩

lemma *map_filter_snd_map_filter*: *List.map_filter* (λ(*a*, *b*). *if P b then None else Some* (*f a b*)) *xs* =
map (λ(*a*, *b*). *f a b*) (*filter* (λ*x*. ¬ *P* (*snd x*)) *xs*)
⟨*proof*⟩

lemma *mk_db_code_alist*:
mk_db t = *Assoc_List_Mapping* (*clearjunk t*)
⟨*proof*⟩

lemma *mk_db_code*[*code*]:

mk_db t = Mapping.of_alist (List.map_filter (λ(p, X). if X = {} then None else Some (p, [X])) (AList.clearjunk t))
⟨*proof*⟩

declare [[*code drop*: *New_max_getIJ_genericJoin New_max_getIJ_wrapperGenericJoin*]]
declare *New_max.genericJoin.simps*[*folded remove_Union_def*, *code*]
declare *New_max.wrapperGenericJoin.simps*[*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.