

# Translation Mechanism for the LogicGuard Abstract Language

Temur Kutsia      Wolfgang Schreiner

RISC, Johannes Kepler University Linz

`{schreine,kutsia}@risc.jku.at`

October 23, 2012

## Abstract

The LogicGuard project aims at developing a specification and verification formalism for runtime network monitoring based on predicate logic. This report describes the translation mechanism of the LogicGuard abstract language (LGAL). The translator accepts a formula that specifies the monitor and produces a program in functional style, seen as a runtime monitor.

## 1 Brief Overview

The goal of the LogicGuard project is to investigate to what extent classical predicate logic formulas are suitable as the basis for the specification and efficient runtime verification of system runs. The specific focus is on computer and network security, concentrating on predicate logic specifications of security properties of network traffic. In the previous report [1], we described the syntax and semantics of the the LogicGuard Abstract Language (LGAL). LGAL represents a fragment of classical predicate logic. It has four-valued semantics, which corresponds to our intuition behind monitoring: A property being monitored over the given stream can be either true, false, or the monitoring can be interrupted because of an error. Yet another possibility is that, at the given time point, it is not known whether the property is true or false or whether an error will occur.

In this report we proceed further, describing the translation mechanism, which is supposed to automatically generate runtime monitors from the specifications. We do not give here the formal definitions, they can be found in the appendix. Rather, we try to informally explain what the mechanism is supposed to do.

We start the explanation by illustrating a single step in runtime network monitoring shown in Figure 1. Note that there is a difference from the basic model of runtime network monitoring described in [1]: Here we allow multiple global streams, called the external streams, and initialize some local, internal streams outside the monitor. (The latter does not forbid to create internal streams inside the monitor, though.)

The network traffic is modeled by the external streams. At each position, each stream contains a message, which is a pair  $(t, v)$  of the time stamp  $t$  and the value  $v$ . (Some of those streams can be finite.) The messages in the streams are linearly ordered according to the time stamp. In monitoring, not only the external streams, but also some internally created streams may be needed. The goal of the translation is to make the (modified version of the) basic model “executable” by translating the *monitor specification* into a program, which we call here the *runtime monitor*.

The monitor is described by its specification  $M = \text{monitor } XP_1 \text{ in } SID_1, \dots, XP_n \text{ in } SID_n : F$ . The monitoring formula  $F$  is interpreted over the streams  $SID_1, \dots, SID_n$ , which refer to external or internal streams. Those streams are created and expanded outside of  $M$ . However,  $F$  may also

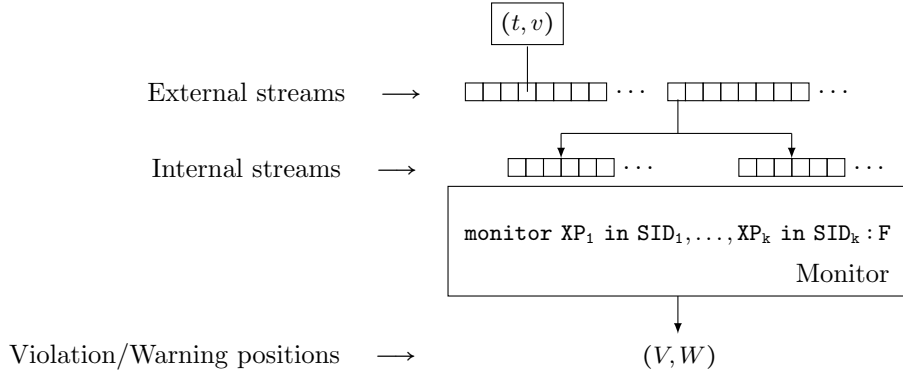


Figure 1: Monitoring Step.

contain some other internal streams, which are created and expanded there (i.e., inside  $M$ ). The monitoring positions  $XP$ 's refer to the positions in the streams denoted by the corresponding  $SID$ 's.

Each time, we take a new message from one of the external streams (we say that a new message *arrives* to one of the external streams, which gets updated by this message), update the internal streams, and evaluate the monitoring formula on the updated external and internal streams. The result of the evaluation can be either true ( $\mathbf{t}$ ), false ( $\mathbf{f}$ ), error ( $\perp_{\mathbf{F}}$ ), or the unknown value ( $?_{\mathbf{F}}$ ). Then the verification system should report violations when the monitored formula evaluates to  $\mathbf{f}$ , and warnings when the monitored formula evaluates to  $?_{\mathbf{F}}$ . Violations are warnings are just positions in the streams  $SID_1, \dots, SID_n$ , which show the values of the variables  $XP_1, \dots, XP_n$ . In the figure, the sets of violations and warnings are denoted respectively by  $V$  and  $W$ .

Note that when a message arrives, the runtime monitor might not be able to determine the truth value of the monitoring formula: There might be simply not enough information on the streams. In this case, the runtime monitor waits for the next message to come.

The behavior of the runtime monitor can be described by executing the monitoring step at each message which arrives to the external streams, where the step itself performs the following actions: When a message arrives, try to evaluate the monitoring formula. If the answer is  $\perp_{\mathbf{F}}$ , abort the entire evaluation. If the answer is  $\mathbf{f}$  or  $?_{\mathbf{F}}$ , report the position and wait for the next message. If the answer is  $\mathbf{t}$  or if we can not get the answer at all, wait for the next message. This top-level control on the evaluation of the monitoring step in the translation is given by the pseudocode below. The translation of monitor specifications into runtime monitors can be found in the appendix.

```

...
state ← (stream-environment, monitoring-formula)
env-fun-pred ← environment-for-external-functions-and-predicates
loop
  receive (message, stream-id)
  (violations, warnings, state) =
    monitoring-step(message, stream-id, state, env-fun-pred)
  report violations
  report warnings

```

The loop is, in general, infinite. It only terminates if there are no new messages coming (i.e., if the external streams are finite), or when the monitoring formula does not depend on the input streams. It may break if the monitoring step raises error.

The monitoring formula expresses a safety property, if every position for which the formula evaluates to false or unknown, belongs to some finite prefix of the input streams. Correctness of the

translation can be formulated, rather informally, as the soundness and completeness statements below, where the latter states completeness with respect to safety properties:

**Soundness:** If the monitor reports a position in an input stream as a violation (or warning), then the truth value of the monitoring formula is false (resp. unknown).

**Completeness:** If the monitoring formula expresses a safety property and its truth value is false (resp. unknown) at some position, then this position will eventually be reported as a violation (resp. warning) position during the monitoring loop.

This report describes the current state of the work-in-progress on the translation mechanism for LGAL. The translator is represented as a function whose definition is given in the appendix. Efficiency issues such as, e.g., history pruning, are not addressed. Formulas are evaluated sequentially, from left to right, in a lazy manner. Proving the correctness property remains as a future work.

## Acknowledgments

The project “LogicGuard: The Efficient Checking of Time-Quantified Logic Formulas with Applications in Computer Security” is sponsored by the FFG BRIDGE program, project No. 832207. The authors thank the project partner companies: SecureGuard GmbH and RISC Software GmbH.

## References

- [1] Temur Kutsia and Wolfgang Schreiner. LogicGuard Abstract Language. RISC Report Series 12-08, Research Institute for Symbolic Computation (RISC), University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, 2012.

```

1
2
3
4
5 =====
6 1. Translation of the Monitor
7 =====
8
9 TYPES
10 - - - - -
11
12 =====
13 STATE
14 =====
15 State = EnvStream × Step[MK] × ℙ(PotentialViolation)
16
17 =====
18 ENVIRONMENTS
19 =====
20 Present = Message × SID
21           × EnvStreams × EnvFormula × EnvPosition × EnvValue
22           × EnvFunctionsPredicates
23
24 EnvStream = ExtEnv × IntEnv
25
26 ExtEnv = ExSID → StreamInfo
27
28 IntEnv = InSID → StreamInfo
29
30 EnvFormula = XF → Step[MK]
31
32 EnvPosition = XP → Step[Position ∪ {?P}]
33
34 EnvValue = XV → Step[Value ∪ {?V}]
35
36 =====
37 BINDINGS
38 =====
39 Binder = Present → ((Present × Binder) ∪ {⊥E})
40
41 =====
42 STREAMS
43 =====
44 StreamInfo = StreamPast × StreamFuture
45
46 StreamPast = Message* × Nat
47
48 StreamFuture = NIL + ?S + Stream + External
49
50 Stream = Present → ⊥S + StreamInfo
51
52 =====
53 STEP
54 =====
55 Step[T] = Present → Answer[T]
56
57 Answer[T] = T + Step[T]
58
59 =====
60 VIOLATIONS, WARNINGS
61 =====
62 PotentialViolation = ((Position × SID)+ × Step[MK])
63
64 Violation = (Position × SID)+
65
66 Warning = (Position × SID)+
67
68 Message = Time × (Value ∪ {?V})
69
70 MessageStream = ℙ(Message) × Stream
71
72 MonitoringVarsSIDs = (XP × SID)*
73 // Monitoring variables and corresponding streams
74

```

```

75 =====
76 MK, VALUES, POSITIONS, CONSTRAINT DOMAINS, RANGES, ETC
77 =====
78 Bool = {true,false}
79 MK = Bool u {?F,⊥F}
80
81 Position? = Position u {?P}
82 Position⊥ = Position u {⊥P}
83 Position⊥? = Position u {⊥P,?P}
84
85 Value? = Value u {?V}
86 Value⊥ = Value u {⊥V}
87 Value⊥? = Value u {⊥V,?V}
88
89 VPS = Value? + Position? + Stream
90
91 ConstrDom = ⊥E + PresentMK
92 PresentMK = Present × MK
93
94 Range = Step[Position⊥?]
95         × {"<","=<"} × XP × {"<","=<"}
96         × Step[Position⊥? u {∞}]
97
98 =====
99 DATA CONSTRUCTORS:
100 =====
101 isMK: MK → MK
102
103 isValue: Value → Value
104 isValue⊥? : Value⊥? → Value⊥?
105
106 isPosition: Position → Position
107 isPosition⊥?: Position⊥? → Position⊥?
108
109 isVPS: VPS → VPS
110 =====
111 -----
112
113 Monitor := ExSID1 ... ExSIDn : M
114           | ExSID1 ... ExSIDn, let InSID1 = TS1 ... InSIDn = TSn : M
115
116 M := monitor XP1 in SID1, ..., XPn in SIDn : F
117
118 SID := ExSID | InSID
119 ExSID := XS
120 InSID := XS
121 -----
122
123 Tr[[Monitor]] : State × InSID* × ℙ(MonitoringVarsSIDs)
124
125 // Initializes the external and internal stream environments mapping
126 // the stream ids to the empty streams, translates the monitoring
127 // formula with free occurrences of monitored position variables, and
128 // creates the initial state consisting of the initial stream
129 // environment, translated monitoring formula, and the empty set
130 // of potential violations. Together with the initial state, it returns
131 // the sequence of internal stream ids and the set of pairs of those
132 // monitoring variables and the corresponding stream ids, which
133 // occur in the monitored formula.
134
135 Tr[[Monitor]] =
136   let (ExtID1,...,ExtIDn, let IntID1 = TS1,...,IntID_m = TS_m, M) = Monitor :
137   let ext_env = {ExtID1 ↦ (ε,0,External), ..., ExtIDn ↦ (ε,0,External)} :
138   let s1 = Tr[[TS1]],..., s_m = Tr[[TS_m]] :
139   let int_env = {IntID1 ↦ (ε,0,s1), ..., IntID_m ↦ (ε,0,s_m)} :
140   let stream_env = ext_env × int_env :
141   let (monitor XP1 in SID1, ..., XPk in SIDk : F) = M :
142   let f = Tr[[F]] :
143   let int_ids = (IntID1,...,IntID_m) :
144   let monitoring_vars_ids =
145     select_vars_occurring_in_formula((XP1,SID1)...,(XPk,SIDk),F) :
146   let potential_violations = ∅ :
147   let initial_state = (stream_env,f,potential_violations) :
148   (initial_state,int_ids,monitoring_vars_ids)

```

```

149
150 -----
151 select_vars_occurring_in_formula :
152     MonitoringVarsSIDs × F → MonitoringVarsSIDs
153 -----
154
155 select_vars_occurring_in_formula((XP1,SID1)...,(XPk,SIDk),F) =
156     { (XP,SID) | (XP,SID) ∈ {(XP1,SID1)...,(XPk,SIDk)} and XP occurs freely in F }
157
158 // Requires an auxiliary function for checking free occurrences of a variable
159 // in a formula
160
161 -----
162 monitoring_step :
163     (Message + NIL + ?S) × ExSID × State × InSID* × ℙ(MonitoringVarsSIDs)
164         × EnvFunctionsPredicates
165     → (ℙ(Violation) × ℙ(Warnings) × State) + MK
166 -----
167
168 // Performs one step of the monitoring process. Recieves a message
169 // from an external stream, updates the stream environment based
170 // on the obtained message, checks whether the potential violation
171 // steps become real violations/warnings, evaluates the monitoring
172 // formula getting more violation/warning/potential violation positions,
173 // and, finally, returns either
174 // (a) a triple of whose first component is all violation position-sid pairs,
175 //     the second component is all warning position-sid pairs,
176 //     and the third component is the new state consisting of the updated
177 //     stream environment, monitoring formula, and all potential violation
178 //     position-step pairs collected together.
179 // or
180 // (b) a definite value true, false, ?F, or ⊥F. This eventually happens
181 //     when the set of monitoring variables occurring in the monitoring formula
182 //     is empty.
183
184
185 monitoring_step(message,sid,state,int_ids,mon_vars_ids,envfp) =
186     let (stream_env,f,pot_violations) = state :
187     let (ext_env,int_env) = stream_env :
188     let (new_stream_env,ids_of_updated_streams) =
189         update_stream_environment(stream_env,message,sid,envfp,int_ids) :
190     if new_stream_env = ⊥E then
191         ⊥F
192     let envf = envp = envv = EmptyEnvironment :
193     let present = (message,sid,new_stream_env,envf,envp,envv,envfp) :
194     if mon_vars_ids = ∅ then
195         let f0 = f(present)
196         case f0 of
197             isMK(mk) : f0
198             isStep(st) : (∅,∅,(stream_env,f,pot_violations))
199     else
200     let v = check_potential_violations(pot_violations,present) :
201     if v = ⊥ then
202         ⊥F
203     else
204     let (violations,warnings,pot_violations0) = v :
205     let v1 = generate_and_evaluate_instances(f,present,
206         mon_vars_ids,ids_of_updated_streams,int_ids) :
207     if v1 = ⊥ then
208         ⊥F
209     else
210     let (violations1,warnings1,pot_violations1) = v1 :
211     let new_violations = violations u violations1 :
212     let new_warnings = warnings u warnings1 :
213     let new_pot_violations = pot_violations0 u pot_violations1 :
214     let new_state = (new_stream_env,f,new_pot_violations) :
215     (new_violations,new_warnings,new_state)
216
217 -----
218 generate_and_evaluate_instances :
219     Step[MK] × Present × ℙ(MonitoringVarsSIDs) × ℙ(SID) × InSID*
220     → (ℙ(Violation) × ℙ(Warnings) × ℙ(PotentialViolation)) u {⊥}
221 -----
222

```

```

223 // Generates instances of the monitor, obtained by evaluating
224 // the monitoring formula at present, with the addition that
225 // the monitored variables in the monitoring formula are
226 // instantiated with the combination
227 // of positions from the updated and non-updated streams.
228 // The instances with the monitoring formula evaluated to false
229 // are collected in violations, those with ?F - in warnings,
230 // and those that are steps - in potential violations (potvals).
231
232
233 generate_and_evaluate_instances(f,present,monitoring_vars_ids,
234                               ids_of_updated_streams,int_ids) =
235   let {(xp1,sid1,r11,r12),...,(xpk,sidk,rk1,rk2)} =
236     get_ranges(present,monitoring_vars_ids,ids_of_updated_streams,int_ids) :
237
238 // The value of instances below is ∅ if one of the SIDs is the empty stream.
239
240 let instances = {((p1,SID1),...,(pk,SIDk),f(present[X1↦p1,...,XPk↦pk]) |
241                 r11 ≤ p1 ≤ r12, ..., rk1 ≤ pk ≤ rk2 and
242                 f(present[X1↦p1,...,XPk↦pk]) ≠ true}
243 If contains_error(instance) = true then
244   ⊥
245 else
246   let violations = {((p1,SID1),...,(pk,SIDk) |
247                   ((p1,SID1),...,(pk,SIDk),false) ∈ instances} :
248   let warnings = {((p1,SID1),...,(pk,SIDk) |
249                  ((p1,SID1),...,(pk,SIDk),?F) ∈ instances} :
250   let potviols = instances \ (violations ∪ warnings) :
251   (violations,warnings,potviols)
252
-----
253
254 get_ranges : Present × ℙ(MonitoringVarsSIDs) × ℙ(SID) × InSID*
255             → ℙ(XP × SID × Position × Position)
-----
256
257
258 // Collects ranges for position variables in updated and non-updated
259 // streams. In the updated streams the collected positions range
260 // from the 1 + maximal position before updating, to the maximal
261 // position after updating. In non-updated streams the collected
262 // positions are all the positions in the stream.
263
264
265 get_ranges(present,monitoring_vars_ids,ids_of_updated_streams,int_ids) =
266   let ext_env = (present↵3)↵1 :
267   let int_env = (present↵3)↵2 :
268   let ranges_for_updated = {(xp,sid,p1,p2) |
269                           (xp,sid) ∈ monitoring_vars_ids,
270                           sid ∈ ids_of_updated_streams,
271                           p1 = if sid ∈ int_ids then
272                               let number_of_new_messages =
273                                 (int_env(sid)↵1)↵2 :
274                                 maximal_position(int_env(sid)↵1)-
275                                   - number_of_new_messages + 1,
276                               else
277                                 let number_of_new_messages =
278                                   (ext_env(sid)↵1)↵2 :
279                                   maximal_position(ext_env(sid)↵1)-
280                                     - number_of_new_messages + 1,
281                                   maximal_position(ext_env(sid)↵1)
282                           p2 = if sid ∈ int_ids then
283                               maximal_position(int_env(sid)↵1)
284                           else
285                               maximal_position(ext_env(sid)↵1)}
286   let ranges_for_non_updated = {(xp,sid,p1,p2) |
287                               (XP,sid) ∈ monitoring_vars_ids,
288                               sid ∉ ids_of_updated_streams,
289                               p1 = 0,
290                               p2 = if sid ∈ int_ids then
291                                   maximal_position(int_env(sid)↵1)
292                               else
293                                   maximal_position(ext_env(sid)↵1)}
294   let ans = ranges_for_updated ∪ ranges_for_non_updated :
295   ans
296

```

```

297 -----
298 maximal_position : StreamInfo → Nat
299 -----
300
301 maximal_position(info) =
302   len(info↑1)-1
303
304 -----
305 len : Message* → Nat
306 -----
307 len is a function that takes a sequence of messages and
308 returns its length
309
310 -----
311 contains_error :  $\mathbb{P}((\text{Position} \times \text{SID})^+ \times (\text{false} + ?F + \perp F + \text{Step}[\text{MK}])))$ 
312                → Bool
313 -----
314
315 contains_error(set) =
316   if set =  $\emptyset$  then
317     false
318   else
319     let elem = select_an_element(set) :
320       let (pos_sid_pairs,b) = elem :
321         if b =  $\perp F$  then
322           true
323         else
324           let new_set = delete_an_element(set,elem) :
325             contains_error(new_set)
326
327 -----
328 update_stream_environment :
329   EnvStreams × (Message + NIL + ?S) × ExSID
330             × EnvFunctionsPredicates × InSID*
331   → (EnvStreams ×  $\mathbb{P}(\text{SID})$ ) ∪ { $\perp E$ }
332 -----
333
334 update_stream_environment(stream_env,msg,sid,envfp,int_ids) =
335   let (ext_env,int_env) = stream_env :
336     let old_past = ext_env(sid)↑1 :
337       let (messages,lastlenght) = old_past :
338         let new_stream =
339           case msg of
340             NIL           : (messages,0,NIL)
341             ?S            : (messages,0,?S)
342             isMessage(m) : (messages||m,1,External)
343         let new_ext_env = ext_env[sid ↦ new_stream] :
344           let new_stream_env=(new_ext_env,int_env) :
345             let e = EmptyEnvironment :
346               let present = (msg,sid,new_stream_env,e,e,e,envfp) :
347                 let ints = update_int_stream_env(present,int_ids,e, $\emptyset$ ) :
348                   if ints =  $\perp E$  then
349                      $\perp E$ 
350                   else
351                     let (new_int_env,updated_int_ids) = ints :
352                       let updated_ids = {sid}∪ updated_int_ids :
353                         let updated_stream_env = (new_ext_env,new_int_env) :
354                           (updated_stream_env,updated_ids)
355
356 -----
357 update_int_stream_env :
358   Present × InSID* × IntEnv ×  $\mathbb{P}(\text{SID})$  → (IntEnv ×  $\mathbb{P}(\text{SID})$ ) ∪ { $\perp E$ }
359 -----
360
361 update_int_stream_env(present, int_ids, env, ids_of_updated_streams) =
362   if int_ids =  $\varepsilon$  then
363     (env,ids_of_updated_streams)
364   else
365     let (id1,rest) = int_ids :
366       let old_ext_env = (present↑3)↑1 :
367         let old_int_env = (present↑3)↑2:
368           let (str_past,str_future) = old_int_env(id1) :
369             case str_future of
370               NIL :

```



```

371     let new_env = env[id1 ↦ (str_past, str_future)] :
372     update_int_stream_env(present, rest, new_env, ids_of_updated_streams)
373   ?S :
374     let new_env = env[id1 ↦ (str_past, str_future)] :
375     update_int_stream_env(present, rest, new_env, ids_of_updated_streams)
376   isStream(s) :
377     let (past_messages, lastlength) = str_past :
378     let s_streaminfo = s(present) :
379     let (s_current_messages, s_future) = new_streaminfo :
380         // s_current_messages has the form of streampast:
381         // Message* × Nat
382     case s_future of
383     ⊥S : ⊥E
384     NIL :
385         let new_past = (past_messages, 0) :
386         let new_future = NIL :
387         let new_env = env[id1 ↦ (new_past, new_future)] :
388         let updated_present =
389             update_present_with_int_stream(present, id1,
390                 new_past, new_future) :
391         update_int_stream_env(updated_present, rest,
392             new_env, {id1}uids_of_updated_streams)
393     ?S :
394         let new_past = (past_messages, 0) :
395         let new_future = ?S :
396         let new_env = env[id1 ↦ (new_past, new_future)] :
397         let updated_present =
398             update_present_with_int_stream(present, id1,
399                 new_past, new_future) :
400         update_int_stream_env(updated_present, rest,
401             new_env, {id1}uids_of_updated_streams)
402     isStream(ms) :
403         let new_past = (past_messages || s_current_messages ↯ 1,
404             s_current_messages ↯ 2) :
405         let new_env = env[id1 ↦ (new_past, s_future)] :
406         let updated_present =
407             update_present_with_int_stream(present, id1,
408                 new_past, s_future) :
409         update_int_stream_env(updated_present, rest,
410             new_env, {id1}uids_of_updated_streams)
411 -----
412 check_potential_violations :
413   ℙ(PotentialViolation) × Present
414   → ℙ(Violation) × ℙ(Warnings) × ℙ(PotentialViolation) u {⊥}
415 -----
416 check_potential_violations_acc(pot_violations, present) =
417   let new_set = {(p_1, SID_1, ..., (p_k, SID_k), f(present)) |
418       (p_1, SID_1, ..., (p_k, SID_k), f) ∈ pot_violations and
419       f(present) ≠ true}
420   if contains_error(new_set) = true then
421     ⊥
422   else
423     let violations = {(p1, SID1), ..., (p_k, SID_k) |
424         ((p1, SID1), ..., (p_k, SID_k), false) ∈ new_set} :
425     let warnings = {(p1, SID1), ..., (p_k, SID_k) |
426         ((p1, SID1), ..., (p_k, SID_k), ?F) ∈ new_set} :
427     let potviols = intances \ (violations u warnings) :
428     (violations, warnings, potviols)
429 -----
430 update_present_with_int_stream :
431   Present × InSID × StreamPast × StreamFuture → Present
432 -----
433 update_present_with_int_stream(message, sid, envs, envf, envp, envv, envfp,
434     id1, new_past, new_future) =
435   let (ext_env, int_env) = envs :
436   let new_envs = (ext_env, int_env[id1 ↦ (new_past, new_future)]) :
437   (message, sid, new_envs, envf, envp, envv, envfp)
438 -----
439 -----
440 -----
441 -----
442 -----
443 -----
444 -----

```

```

445 =====
446 2. Translation of Formulas
447 =====
448
449 TYPES
450 - - - - -
451
452 Tr[[F]] = step
453
454 Input: a formula F
455 Output: a function "step" that takes as input the present situation, i.e.
456
457 message      ... the current message
458 stream_id    ... the current stream name
459 envs         ... stream environment
460 envf         ... formula environment
461 envp         ... position environment
462 envv         ... value environment
463 envfunpr     ... environment for external functions and predicates
464              (mapping from the names of nonlogical function and predicate
465              names to external functions and preducatcs)
466
467 and returns as an answer
468
469 * either true, false,  $\perp F$ , or ?F
470 * or another step to be applied to the next situation.
471 =====
472
473 -----
474 Tr[[F]]: Step[MK]
475
476 F ::= XF | BIND F
477      | true | false | PV(T1,...,Tn)
478      | ~ F | F1 /\ F2 | F1 \/ F2 | F1 => F2 | F1 <=> F2
479      | forall XP in SID with RAN : F | exists XP in SID with RAN: F
480 -----
481
482 Tr[[XF]](message,stream_id,envs,envf,envp,envv,envfunpr) =
483   inAnswer[MK](envf(XF))
484
485 Tr[[BIND : F]] =
486   let f: Binder  $\times$  Step[MK]  $\rightarrow$  Step[MK]
487     f[bind0,f0](present) =
488       let bind1 = bind0(present) :
489         if bind1 =  $\perp E$  then
490           inAnswer[MK]( $\perp F$ )
491         else
492           let f1 : f0(bind1 $\downarrow$ 1) :
493             case f1 of
494               isMK(mk)      : f1
495               isStep[MK](mkstep) : f[bind1,mkstep]
496           : f[Tr[[BIND]],Tr[[F]]]
497
498 Tr[[true]](present) =
499   inAnswer[MK](true)
500
501 Tr[[false]](present) =
502   inAnswer[MK](false)
503
504 Tr[[PV(T1,...,Tn)]] =
505   let f: (Present  $\rightarrow$  VPSn  $\rightarrow$  MK)  $\times$  Step[VPS]n  $\rightarrow$  Step[MK] :
506     f[pv,t1,...,tn](present) =
507       let r1 = t1(present),...,rn = tn(present):
508         if r1 =  $\perp V$  v ... v rn =  $\perp V$  v r1 =  $\perp P$  v ... v rn =  $\perp P$  then
509           inAnswer[MK]( $\perp F$ )
510         else
511           let i1,...,ik be the maximal subsequence of 1,...,n such that
512             ri1 = isVPS(v1)  $\wedge$  ...  $\wedge$  rik = isVPS(vk) :
513           let j1,...,j{n-k} be the subsequence of 1,...,n such that
514             rj1 = isStep[VPS](s1)  $\wedge$  ...  $\wedge$  rj{n-k} = isStep[VPS](s{n-k}) :
515           if k = n then
516             let m = pv(present)(v1,...,vk) :
517               inAnswer[MK](m)
518           else

```

```

519     let g1,...,gk : Step[VPS]
520     g1(present) = inAnswer[VPS](v1),...,
521     gk(present) = inAnswer[VPS](vk) :
522     let ter1,...,tern be the sequence such that for all 1≤i≤n,
523     if i ∈ {i1,...,ik} then
524     teri = gi
525     else
526     teri = ri :
527     inAnswer[MK](f[ter1,...,tern])
528 : f[Tr[[PV]], Tr[[T1]],...,Tr[[Tn]]]
529
530 Tr[~F] =
531 let f: Step[MK] → Step[MK],
532 f[f0](present) =
533 let r = f0(present):
534 case r of:
535 isMK(m):
536 if m = false then
537 inAnswer[MK](true)
538 else if m = true then
539 inAnswer[MK](false)
540 else if m = ?F then
541 inAnswer[MK](?F)
542 else
543 inAnswer[MK](⊥F)
544 isStep[MK](s):
545 inAnswer[MK](f[s])
546 : f[Tr[[F]]]
547
548 Tr[[F1 ∧ F2]] =
549 let f: Step[MK] × Step[MK] → Step[MK],
550 f[f1,f2](present) =
551 let r1 = f1(present):
552 case r1 of
553 isMK(m1):
554 if m1 = ⊥F ∨ m1 = false then
555 r1
556 else
557 let r2 = f2(present):
558 case r2 of
559 isMK(m2) :
560 if m2 = true then
561 r1
562 else
563 r2
564 isStep[MK](s2):
565 inAnswer[MK](f[f1,s2])
566 isStep[MK](s1):
567 inAnswer[MK](f[s1,f2])
568 : f[Tr[[F1]],Tr[[F2]]]
569
570 Tr[[F1 ∨ F2]] =
571 let f: Step[MK] × Step[MK] → Step[MK],
572 f[f1,f2](present) =
573 let r1 = f1(present):
574 case r1 of
575 isMK(m1):
576 if m1 = ⊥F ∨ m1 = true then
577 r1
578 else
579 let r2 = f2(present):
580 case r2 of
581 isMK(m2):
582 if m2 = false then
583 r1
584 else
585 r2
586 isStep[MK](s2)
587 if m1 = false then
588 r2
589 else
590 inAnswer[MK](f[f1,s2])
591 isStep[MK](s1):
592 inAnswer[MK](f[s1,f2])

```

```

593 : f[Tr[[F1]],Tr[[F2]]]
594
595 Tr[[F1 => F2]] =
596   let f: Step[MK] × Step[MK] → Step[MK],
597       f[f1,f2](present) =
598         let r1 = f1(present):
599           case r1 of
600             isMK(m1):
601               if m1 = ⊥F then
602                 r1
603             else if m1 = false then
604               inAnswer[MK](true)
605             else
606               let r2 = f2(present)
607                 case r2 of
608                   isMK(m2):
609                     if m2 = false then
610                       if m1 = true then
611                         false
612                       else
613                         ?F
614                     else
615                       r2
616                   isStep[MK](s2):
617                     if m1 = true then
618                       r2
619                     else
620                       inAnswer[MK](f[f1,s2])
621                   isStep[MK](s1):
622                     inAnswer[MK](f[s1,f2])
623   : f[Tr[[F1]],Tr[[F2]]]
624
625 Tr[[F1 <=> F2]] =
626   let f: Step[MK] × Step[MK] → Step[MK],
627       f[f1,f2](present) =
628         let r1 = f1(present):
629           case r1 of
630             isMK(m1):
631               if m1 = ⊥F then
632                 r1
633             else if m1 = ?F then
634               r1
635             else
636               let r2 = f2(present):
637                 if m1 = true then
638                   r2
639                 else
640                   case r2 of
641                     isMK(m2):
642                       if m2 = true then
643                         inAnswer[MK](false)
644                       else if m2 = false then
645                         inAnswer[MK](true)
646                       else
647                         r2
648                   isStep[MK](s2):
649                     inAnswer[MK](f[false,s2])
650                   isStep[MK](s1):
651                     inAnswer[MK](f[s1,f2])
652   : f[Tr[[F1]],Tr[[F2]]]
653
654 // Type checker should guarantee that the positions in RAN
655 // are positions in the stream SID
656
657 Tr[[forall XP in SID with RAN : F]] =
658   all_ex["forall"](XP,TS,RAN,F)
659
660 Tr[[exists XP in SID with RAN : F]] =
661   all_ex["exists"](XP,TS,RAN,F)
662
663 -----
664 all_ex : {"forall", "exists"}
665         → XP × SID × RAN × F
666         → Step[MK]

```

```

667 -----
668 // all_ex first fixes the boundaries of the range
669 // then computes the truth value of the quantified formula.
670 // If boundaries are unknown or error position, it returns
671 // unknown and error truth values, respectively.
672 // If a boundary is a step, all_ex returns a step.
673 // If both boundaries are definite positions (finite or infinite)
674 // it tries to compute the truth value.
675
676 all_ex[forall_exists](XP,SID,RAN,F) =
677   let ae : {"forall", "exists"}
678         → XP × SID × (Present → Range u {?P,⊥P}) × F
679         → Step[MK]
680   ae[al_ex](xp,sid,ran,formula)(present) =
681     let r = ran(present) :
682       if r = ?P then
683         inAnswer[MK](?F)
684       else if r = ⊥P then
685         inAnswer[MK](⊥F)
686       else
687         let (tp1,pp1,xp0,pp2,tp2) = r :
688           if xp0 ≠ xp then
689             inAnswer[MK](⊥F)
690           else case tp1 of
691             isStep[Position⊥?](s1) :
692               let ran1 = (s1,pp1,xp0,pp2,tp2) :
693                 inAnswer[MK](ae[forall_exists](xp,sid,ran1,formula))
694             isPosition(pos1) :
695               case tp2 of
696                 isStep[Position⊥?](s2) :
697                   let ran2 = (pos1,pp1,xp0,pp2,s2) :
698                     inAnswer[MK](ae[forall_exists](xp,sid,ran2,formula))
699                 isPosition(pos2) :
700                   let (from,to0) =
701                     precise_range_boundaries(pos1,pp1,pp2,pos2) :
702                   let default =
703                     if al_ex = "forall" then
704                       true
705                     else
706                       false :
707                   forall_exists_pos_pos[default,default]
708                     (from,xp,to,sid,formula)(present)
709 : ae[forall_exists](XP,SID,Tr[[RAN]],F)
710
711 -----
712 forall_exists_pos_pos : Boolu{?F} × Bool
713                       → Position × XP × Position × SID × F
714                       → Step[MK]
715 -----
716
717 forall_exists_pos_pos[cur,def](from,xp,to,sid,fml)(present) =
718   if to > from then
719     inAnswer[MK](cur) : // does not match semantics
720   else
721     let streampast = get_stream_past(sid,present) :
722       if streampast = ⊥S then
723         inAnswer[MK](⊥F)
724       else
725         let max_pos = maximal_position(streampast) :
726           if max_pos < from then
727             inAnswer[MK](forall_exists_pos_pos[cur,def](from,xp,to,sid,fml))
728           else
729             if max_pos < to then
730               let upper = max_pos :
731                 let flag = "incomplete" :
732             else
733               let upper = to :
734                 let flag = "complete" :
735             let f =
736               forall_exists_history[cur,def](from,xp,upper,fml)(present) :
737             case f of
738               isMK(mk) :
739                 if flag = "complete" then
740                   mk

```

```

741         else if mk = ⊥F v mk = -def then
742             mk
743         else
744             inAnswer[MK](forall_exists_pos_pos[mk,def]
745                 (upper, xp, to, fml))
746         isPositionBool?(pos_b) :
747             let (p,b) = pos_b :
748                 inAnswer[MK](forall_exists_pos_pos[b,def](p, xp, to, fml))
749
750 -----
751 forall_exists_history : Boolu{?F} × Bool
752     → Position × XP × Position × Step[MK]
753     → Present
754     → MK + PositionBool?
755
756 PositionBool? = Position × Bool u {?F}
757 -----
758
759 forall_exists_history[current,default](from, xp, to, formula)(present) =
760     if from > to then
761         inMK(current)
762     else
763         let f = formula(present[xp→from]) :
764             case f of
765                 isMK(mk) :
766                     if mk = ⊥F v mk = -default then
767                         inMK(mk)
768                     else if mk = default then
769                         forall_exists_history[current,default]
770                             (from+1, xp, to, formula)(present)
771                     else // mk = ?F
772                         forall_exists_history[mk,default]
773                             (from+1, xp, to, formula)(present)
774                 isStep[MK](st) :
775                     let pb = (from,current) :
776                         inPositionBool?(pb)
777
778 -----
779 precise_range_boundaries :
780     Position × {"≤", "<"} × {"≤", "<"} × Position
781     → Position × Position
782 -----
783
784 precise_range_boundaries(pos1, pp1, pp2, pos2) =
785     let from =
786         if pp1 = "≤" then
787             pos1
788         else
789             pos1 + 1
790     let to0 =
791         if pp2 = "≤" v pos2 = "∞" then
792             pos2
793         else
794             pos2 - 1
795     : (from, to)
796
797 -----
798 Tr[[TP]]: Step[Position⊥?]
799 TP ::= XP | BIND : TP
800     | 0 | increment(SID⊙TP, N) | decrement(SID@TP, N)
801     | max XP in SID with RAN : F | min XP in SID with RAN : F
802 -----
803
804 Tr[[XP]](message, stream_id, history, envf, envp, envv, envfunpr) =
805     inAnswer[Position⊥?](envp(XP))
806
807 Tr[[ BIND : TP ]] =
808     let f: Binder × Step[Position⊥?] → Step[Position⊥?]
809         f[bind0, f0](present) =
810             let bind1 = bind0(present) :
811                 if bind1 = ⊥E then
812                     inAnswer[Position⊥?](⊥P)
813                 else
814                     let p : f0(bind1↓1) :

```

```

815         case p of
816             isPosition1?(p1)           : p1
817             isStep[Position1?](pstep) : f[bind1,pstep]
818     : f[Tr[[BIND]],Tr[[TP]]]
819
820 Tr[[0]] = inAnswer[Position](0)
821
822 Tr[[increment(SID⊙TP,N)]] =
823     let f: SID × TP × Value → Step[Position1?] :
824         f[sid0, pos0, v0](present) =
825             let t = time_or_content(sid0,pos0,"time")(present) :
826                 case t of
827                     isValue1?(time)      :
828                         if time = ⊥V then
829                             inAnswer[Position1?](⊥P)
830                         else if time = ?V then
831                             inAnswer[Position1?](?P)
832                         else
833                             let streampast = get_stream_past(sid0,present) :
834                                 if streampast = ⊥S then
835                                     inAnswer[Position1?](⊥P)
836                                 else
837                                     let max_pos = maximal_position(streampast) :
838                                         let (tmax,_) = extract_message(streampast,max_pos,0) :
839                                             if tmax ≥ time + v0 then
840                                                 let p1 = min p :
841                                                     extract_message(streampast,p,0)⊥1 ≥ time + v0 :
842                                                         inAnswer[Position1?](p1)
843                                                 else
844                                                     inAnswer[Position1?](f(sid0, pos0, v0))
845                                     isStep(Value1?)(v) :
846                                         inAnswer[Position1?](f(sid0, pos0, v0))
847             : f(SID,TP,Tr[[N]])
848
849 Tr[[decrement(SID⊙TP,N)]] =
850     let f: SID × Step[Position1?] × Value → Step[Position1?] :
851         f[sid0, pos0, v0](present) =
852             let t = time_or_content(stream0,pos0,"time")(present) :
853                 case t of
854                     isValue1?(time)      :
855                         if time = ⊥V then
856                             inAnswer[Position1?](⊥P)
857                         else if time = ?V then
858                             inAnswer[Position1?](?P)
859                         else
860                             let tmin = time - v0 :
861                                 if tmin < 0 then
862                                     inAnswer[Position1?](⊥P)
863                                 else
864                                     let streampast = get_stream_past(sid0,present) :
865                                         if streampast = ⊥S then
866                                             inAnswer[Position1?](⊥P)
867                                         else
868                                             let p1 = min p :
869                                                 extract_message(streampast,p,0)⊥1 ≥ time - v0 :
870                                                     inAnswer[Position1?](p1)
871                                     isStep(Value1?)(v) :
872                                         inAnswer[Position1?](f(sid0, pos0, v0))
873             : f(SID,Tr[[TP]],Tr[[N]])
874
875
876 Tr[[min XP in SID with RAN : F]] =
877     min_max["min"](XP,TS,RAN,F)
878
879 Tr[[max XP in SID with RAN : F]] =
880     min_max["max"](XP,TS,RAN,F)
881
882 -----
883 min_max : {"min", "max"}
884         → XP × SID × RAN × F
885         → Step[Position1?]
886 -----
887
888 min_max[min_or_max](XP,SID,RAN,F) =

```

```

889 let minmax : {"min", "max"}
890           → XP × SID × (Present → Range U {?P,⊥P}) × F
891           → Step[Position⊥?]
892 minmax[mm](xp,sid,ran,formulas)(present) =
893   let r = ran(present) :
894   if r = ?P then
895     inAnswer[Position⊥?](?P)
896   else if r = ⊥P then
897     inAnswer[Position⊥?](⊥P)
898   else
899     let (tp1,pp1,xp0,pp2,tp2) = r :
900     if xp0 ≠ xp then
901       inAnswer[Position⊥?](⊥P)
902     else case tp1 of
903       isStep[Position⊥?](s1) :
904         let ran1 = (s1,pp1,xp0,pp2,tp2) :
905         inAnswer[Position⊥?](minmax[mm](xp,sid,ran1,formula))
906       isPosition(pos1) :
907         case tp2 of
908           isStep[Position⊥?](s2) :
909             let ran2 = (pos1,pp1,xp0,pp2,s2) :
910             inAnswer[Position⊥?](minmax[mm](xp,sid,ran1,formula))
911           isPosition(pos2) :
912             let (from,to) =
913               precise_range_boundaries(pos1,pp1,pp2,pos2) :
914             let default = ?P:
915               min_max_pos_pos[mm,default]
916               (from,xp,to,sid,formula)(present))
917 : minmax[min_or_max](XP,SID,Tr[[RAN]],F)
918
919 -----
920 min_max_pos_pos : {"min","max"} × PositionU{?P}
921                 → Position × XP × Position × SID × F
922                 → Step[Position⊥?]
923 -----
924
925 min_max_pos_pos[min_or_max,p](from,xp,to,sid,formula)(present) =
926   if from > to then
927     inAnswer[Position⊥?](p)
928   else
929     let streampast = get_stream_past(sid,present) :
930     if streampast = ⊥S then
931       inAnswer[Position⊥?](⊥P)
932     else
933       let max_pos = maximal_position(streampast) :
934       if max_pos < from then
935         inAnswer[Position⊥?]
936         (min_max_pos_pos[min_or_max,p](from,xp,to,sid,formula))
937       else
938         if max_pos < to then
939           let upper = max_pos :
940           let flag = "incomplete" :
941         else
942           let upper = to :
943           let flag = "complete" :
944         let pos = min_max_history[min_or_max,p](from,xp,upper,sid,formula)
945           (present):
946         case pos of
947           isPosition⊥?(pos0) :
948             if flag = "complete" then
949               inAnswer[Position⊥?](pos0)
950             else
951               if pos0 = ⊥P then
952                 inAnswer[Position⊥?](⊥P)
953               else if pos0 = ?P then
954                 inAnswer[Position⊥?]
955                 (min_max_pos_pos[min_or_max,pos0]
956                  (upper,xp,to,sid,formula))
957               else if min_max = "min" then
958                 inAnswer[Position⊥?](pos0)
959               else // min_max = "max" ∧ flag = "incomplete"
960                 inAnswer[Position⊥?]
961                 (min_max_pos_pos[min_or_max,pos0]
962                  (upper,xp,to,sid,formula))

```



```

963         isPositionPos?(pos_pos)
964         let (bound,p) = pos_pos :
965         inAnswer[PositionI?]
966             (min_max_pos_pos[min_or_max,p](bound,xp,to,sid,formula))
967
968 -----
969 min_max_history : {"min","max"} × Position u {?}
970                 → Position × XP × Position × F
971                 → Present
972                 → PositionI? + PositionPos?
973
974 PositionPos? = Position × (Position u {?P})
975 -----
976
977 min_max_history[min_max,pos](from,xp,to,formula)(present) =
978   if from > to then
979     inPositionI?(pos)
980   else
981     let f = formula(present[xp→from]) :
982     case f of
983       isMK(mk) :
984         if mk = ⊥F then
985           inPositionI?(⊥P)
986         else if mk = true then
987           if min_max = "min" then
988             inPositionI?(from)
989           else
990             min_max_history[min_max,from](from+1,xp,to,formula)(present)
991         else // if mk=false v mk=?F
992             min_max_history[min_max,pos](from+1,xp,to,formula)(present)
993     isStep[MK](st) :
994       let pp = (from,pos) :
995       inPositionPos?(pp)
996
997 -----
998 Tr[[TV]]: Step[ValueI?]
999
1000 TV ::= XV | BIND : TV
1001       | SID⊙TP | SID@TP
1002       | FV(TV1,...,TVn) | num XP in SID with RANB : F
1003       | complete combine[TV0, FV] XP in SID with RAN CONSTR until F : TV1
1004 -----
1005
1006 Tr[[XV]](message,stream_id,envs,envf,envp,envv,envfunpr) =
1007   inAnswer[ValueI?](env(XV))
1008
1009 Tr[[ BIND : TV ]] =
1010   let f: Binder × Step[ValueI?] → Step[ValueI?]
1011   f[bind0,f0](present) =
1012     let bind1 = bind0(present) :
1013     if bind1 = ⊥E then
1014       inAnswer[ValueI?](⊥V)
1015     else
1016       let p : f0(bind1↓1) :
1017       case p of
1018         isValueI?(v1)           : v1
1019         isStep[ValueI?](vstep) : f[bind1,vstep]
1020   : f[Tr[[BIND]],Tr[[TV]]]
1021
1022 // Below the type checker should make sure that TP is a position
1023 // in the stream referred by SID
1024
1025 Tr[[SID⊙TP]] =
1026   let tp = Tr[[TP]] :
1027   time_or_content[SID,tp,"time"]
1028
1029 Tr[[SID@TP]] =
1030   let tp = Tr[[TP]] :
1031   time_or_content[SID,tp,"content"]
1032
1033 -----
1034 time_or_content : SID × Step[PositionI?] × {"time", "content"}
1035                 → Step[ValueI?]

```

```

1036 -----
1037
1038 time_or_content[SID,tp,time_content] =
1039   let f: SID × Step[PositionI?] × {"time", "content"}
1040       → Step[ValueI?]
1041   f[sid,tp0,tv](present) =
1042     let streampast = get_stream_past(sid,present)
1043     if streampast = ⊥S then
1044       inAnswer[ValueI?](⊥V)
1045     else
1046       let p = tp0(present) :
1047       case p of
1048         isPositionI?(p0) :
1049           if p0 = ⊥P then
1050             inAnswer[ValueI?](⊥V)
1051           else if p0 = ?P then
1052             inAnswer[ValueI?](?V)
1053           else if maximal_position(streampast) < p0 then
1054             inAnswer[ValueI?](f(sid,tp0,tv))
1055           else
1056             let m = extract_message(streampast,p0) :
1057             let (t,content) = m :
1058             if tv = "time" then
1059               inAnswer[ValueI?](t)
1060             else
1061               inAnswer[ValueI?](content)
1062             isStep[PositionI?](tp1) :
1063             inAnswer[ValueI?](f(sid,tp1,tv))
1064   : f[SID,tp,time_content]
1065 -----
1066
1067 get_stream_past : SID × Present → StreamPast u {⊥S}
1068 -----
1069
1070 get_stream_past(id,present) =
1071   let ext_env = (present↘3)↘1 :
1072   if id ∈ dom(ext_env) then
1073     ext_env(id)
1074   else
1075     let int_env = (present↘3)↘2 :
1076     if id ∈ dom(int_env) then
1077       int_env(id)↘1
1078     else
1079       ⊥S
1080 -----
1081
1082 extract_message : StreamPast × Position → Message
1083 -----
1084
1085 extract_message(streampast,p) =
1086   let (m,rest) = streampast :
1087   if p = 0 then
1088     m
1089   else
1090     let p1 = p - 1
1091     extract_message(rest,p1)
1092 -----
1093 Tr[[FV(T1,...,Tn)]] =
1094   let f: (Present → VPSn → ValueI?) × Step[VPS]n
1095       → Step[ValueI?] :
1096   f[fv,t1,...,tn](present) =
1097     let r1 = t1(present),...,rn = tn(present):
1098     if r1 = ⊥V v ... v rn = ⊥V v r1 = ⊥P v ... v rn = ⊥P then
1099       inAnswer[ValueI?](⊥F)
1100     else
1101       let i1,...,i_k be the maximal subsequence of 1,...,n such that
1102         r_{i1} = isVPS(v1) ∧ ... ∧ r_{i_k} = isVPS(v_k) :
1103       let j1,...,j_{n-k} be the subsequence of 1,...,n such that
1104         r_{j1} = isStep[VPS](s1) ∧ ... ∧ r_{j_{n-k}} = isStep[VPS](s_{n-k}) :
1105       if k = n then
1106         let m = fv(present)(v1,...,v_k) :
1107         inAnswer[ValueI?](m)
1108       else
1109         let g1,...,gk : Step[VPS]

```

```

1110         gi(present) = inAnswer[VPS](v1),...,
1111         gk(present) = inAnswer[VPS](vk) :
1112     let ter1,...,tern be the sequence such that for all 1≤i≤n,
1113         if i ∈ {i1,...,ik} then
1114             teri = gi
1115         else
1116             teri = ri :
1117     inAnswer[Value1?](f[ter1,...,tern])
1118 : f[Tr[[FV]],Tr[[T1]],...,Tr[[Tn]]]
1119
1120 // Type checking should guarantee that the positions in RAN refer to
1121 // the stream SID
1122
1123 Tr[[num XP in SID with RAN : F]] =
1124     let number : XP × SID × Range u {?P,⊥P} × F → Step[Value1?]
1125     number(xp,sid,ran,formula)(present) =
1126         let r = ran(present) :
1127         if r = ?P then
1128             inAnswer[Value1?](?V)
1129         else if r = ⊥P then
1130             inAnswer[Value1?](⊥V)
1131         else
1132             let (tp1,pp1,xp0,pp2,tp2) = r :
1133             if xp0 ≠ xp then
1134                 inAnswer[Value1?](⊥V)
1135             else case tp1 of
1136                 isStep[Position1?](s1) :
1137                     let ran1 = (s1,pp1,xp0,pp2,tp2) :
1138                     inAnswer[Value1?](number(xp,sid,ran1,formula))
1139                 isPosition(pos1) :
1140                     isStep[Position1?](s2) :
1141                         let ran2 = (pos1,pp1,xp0,pp2,s2) :
1142                         inAnswer[Value1?](number(xp,sid,ran1,formula))
1143                 isPosition(pos2) :
1144                     let (from,to) =
1145                         precise_range_boundaries(pos1,pp1,pp2,pos2) :
1146                     let default = 0 :
1147                     num_pos_pos[default](from,xp,to,sid,formula)(present))
1148 : number(XP,SID,Tr[[RAN]],F)
1149
1150 -----
1151 num_pos_pos : Value
1152             → Position × XP × Position × SID × F
1153             → Step[Value1?]
1154 -----
1155
1156 num_pos_pos[v](from,xp,to,sid,formula)(present) =
1157     if from > to then
1158         inAnswer[Value1?](v)
1159     else
1160         let streampast = get_stream_past(sid,present) :
1161         if streampast = ⊥S then
1162             inAnswer[Value1?](⊥V)
1163         else
1164             let max_pos = maximal_position(streampast) :
1165             if max_pos < from then
1166                 inAnswer[Value1?](num_pos_pos[v](from,xp,to,sid,formula))
1167             else
1168                 if max_pos < to then
1169                     let upper = max_pos :
1170                     let flag = "incomplete" :
1171                 else
1172                     let upper = to :
1173                     let flag = "complete" :
1174                 let val = num_history[v](from,xp,upper,formula)(present):
1175                 case val of
1176                     isValue1?(val0) :
1177                         if flag = "complete" then
1178                             inAnswer[Value1?](val0)
1179                     else
1180                         if val0 = ⊥V then
1181                             inAnswer[Value1?](⊥V)
1182                         else // val0 is never ?V. Hence, it is a number here
1183                             inAnswer[Value1?](num_pos_pos[val0])

```

```

1184                                     (upper, xp, to, sid, formula))
1185         isPositionVal?(pos_val) :
1186             let (bound, v1) = pos_val :
1187                 inAnswer[Value1?](num_pos_pos[v1](bound, xp, to, sid, formula))
1188
1189 -----
1190 num_history : Value
1191             → Position × XP × Position × formula
1192             → Present
1193             → Value1? + PositionVal?
1194
1195 PositionVal? = Position × ValueU{?V}
1196 -----
1197
1198 num_history[val](from, xp, to, formula)(present) =
1199     if from > to then
1200         inValue1?(val)
1201     else
1202         let f = formula(present[xp→from]) :
1203             case f of
1204                 isMK(mk) :
1205                     if mk = ⊥F then
1206                         inValue1?(⊥V)
1207                     else if mk = true then
1208                         num_history[val+1](from+1, xp, to, formula)(present)
1209                     else
1210                         num_history[val](from+1, xp, to, formula)(present)
1211                 isStep[MK](st) :
1212                     let pv = (from, v) :
1213                         inPositionVal?(pv)
1214
1215 Tr[complete combine[TV0, FV] XP in SID with RAN CONSTR until F : TV1] =
1216     let f : Step[Value1?] × (Present → VPSn → Value1?)
1217         × XP × SID
1218         × Present → Range U {?P, ⊥P}
1219         × Step[ConstrDom]
1220         × Step[MK]
1221         × Step[Value1?]
1222         → Step[Value1?]
1223     f[v0, fv, xp, sid, r, c, f, v1](present) =
1224     let r_loc = r(present) :
1225         if r_loc = ⊥P then
1226             ⊥V
1227         else if r_loc = ?P then
1228             ?V
1229         else
1230             let (p1, pp1, xp1, pp2, p2) = r_loc :
1231                 if xp1 ≠ xp then
1232                     ⊥V
1233                 else case p1 of
1234                     isStep[Position1?](pos1) :
1235                         let r1 = (pos1, pp1, xp1, pp2, p2) :
1236                             inAnswer[Value1?](f[v0, fv, xp, sid, r1, c, f, v1])
1237                     isPosition1?(pos1) :
1238                         if pos1 = ⊥P then
1239                             ⊥V
1240                         else if pos1 = ?P then
1241                             ?V
1242                     else
1243                         case p2 of
1244                             isStep[Position1?](pos2) :
1245                                 let r1 = (pos1, pp1, xp1, pp2, pos2) :
1246                                     inAnswer[Value1?](f[v0, fv, xp, sid, r1, c, f, v1])
1247                             isPosition1?(pos2) :
1248                                 if pos2 = ⊥P then
1249                                     ⊥V
1250                                 else if pos2 = ?P then
1251                                     ?V
1252                             else
1253                                 let streampast = get_stream_past(sid, present) :
1254                                     if streampast = ⊥S then
1255                                         ⊥V
1256                                 else
1257                                     let r1 = (pos1, pp1, xp1, pp2, pos2) :

```



```

1332         inAnswer[Value⊥V](⊥V)
1333     else if f1 = ?F v f1 = false then
1334         let from1 = from + 1 :
1335         inAnswer[Value⊥?]
1336             (value_comb[xp,sid,constr,
1337                formula,val, combine_values]
1338              (from1, to, newacc))
1339     else
1340         inAnswer[Value⊥V](newacc)
1341
1342 -----
1343 Tr[[TS]]: Stream
1344
1345 TS ::= ES | IS
1346
1347 ES ::= ExSID
1348
1349 IS ::= InSID | BIND : IS
1350     | FS(T1,...,Tn)
1351     | partial combine[TV0, FV] XP in SID with RAN CONSTR until F : TV
1352     | Default for F in 'until F' : false
1353     | construct XP in SID with RAN CONSTR : TV
1354     | construct XP in SID with RAN CONSTR : IS
1355
1356 SID ::= ExSID | InSID
1357 -----
1358
1359 Tr[[XS]](message,stream_id,envs,envf,envp,envv,envfunpr) =
1360     envs(XS)
1361
1362 Tr[[ BIND : IS ]] =
1363     let f: Binder × Stream → Stream
1364         f[bind0,s0](present) =
1365             let bind1 = bind0(present) :
1366             if bind1 = ⊥E then
1367                 ⊥S
1368             else
1369                 let present1 = bind1⊥1 :
1370                 let s1 = s0(present1) :
1371                 case s1 of
1372                     is⊥S : ⊥S
1373                     isStreamInfo(info) :
1374                         let info = (streampast,streamfuture) :
1375                         let (messages,l) = streampast :
1376                         case streamfuture of
1377                             isNIL :
1378                                 info
1379                             is?S :
1380                                 info
1381                             isExternal :
1382                                 f[bind1,info]
1383                             isStream(s) :
1384                                 f[bind1,info]
1385                 : f[Tr[[BIND]],Tr[[IS]]]
1386
1387 Tr[[FS(T1,...,Tn)]] =
1388     let f: (Present → VPSn → Stream) × Step[VPS]n → Stream :
1389     f[fs,t1,...,tn](present) =
1390     let r1 = t1(present),...,rn = tn(present):
1391     if r1 = ⊥V v ... v rn = ⊥V v r1 = ⊥P v ... v rn = ⊥P then
1392         ⊥S
1393     else
1394         let i1,...,i_k be the maximal subsequence of 1,...,n such that
1395             r_{i1} = isVPS(v1) ∧ ... ∧ r_{i_k} = isVPS(v_k) :
1396         let j1,...,j_{n-k} be the subsequence of 1,...,n such that
1397             r_{j1} = isStep[VPS](s1) ∧ ... ∧ r_{j_{n-k}} = isStep[VPS](s_{n-k}) :
1398         if k = n then
1399             (fs(present)(v1,...,v_k))(present)
1400         else
1401             let g1,...,gk : Step[VPS]
1402             g1(present) = inAnswer[VPS](v1),...,
1403             g_k(present) = inAnswer[VPS](v_k) :
1404             let ter1,...,ter_n be the sequence such that for all 1≤i≤n,
1405             if i ∈ {i1,...,i_k} then

```







```

1554                                     combine_values](from,to,acc))
1555 // The call to f above will uses acc and not newacc
1556 // in order not to keep increasing newacc while
1557 // staying within the same range (from,to) as long as f1 is the step.
1558         isMK(b2) :
1559             if b2 = ⊥F then
1560                 inAnswer[StreamPast⊥?](⊥S)
1561             else if b2 ≠ false then
1562                 inAnswer[StreamPast⊥?](newacc)
1563             else if from ≥ to then
1564                 inAnswer[StreamPast⊥?](newacc)
1565             else
1566                 let from1 = from + 1 :
1567                 inAnswer[StreamPast⊥?]
1568                     (streampast_comb[xp,c,formula,tv,
1569                     combine_values](from1,to,newacc))
1570
1571 -----
1572 combine and join :
1573     StreamPast
1574     × Value u {?V}
1575     × (Value u {?V} × Value u {?V} → Value⊥?)
1576     → StreamPast u {⊥V}
1577 -----
1578
1579 combine and join(past, v, combine_values) =
1580 let cur_time = (past⊥1)⊥1 :
1581 let (messages,l) = past :
1582 if v = ?V then
1583     let newmessages = messages||(cur_time, v) :
1584     (newmessages,1)
1585 else
1586     let k = max i : extract_message(s, i, 0) ≠ (_,?V) :
1587     let (tk,vk) = extract_message(s, k, 0) :
1588     let newval = combine_values(vk, v) :
1589     if newval = ⊥V then
1590         ⊥S
1591     else
1592         let newmessages = messages||(cur_time, v) :
1593         (newmessages,1)
1594
1595 Tr[[construct XP in SID with RAN CONSTR : TS]] =
1596 let f : XP × SID
1597     × Present → Range u {?P,⊥P}
1598     × Step[ConstrDom]
1599     × Stream
1600     × StreamInfo*
1601     → Stream
1602 f[xp,sid,r,c,ts,strinfos_acc](present) =
1603 let r_loc = r(present) :
1604 let past = (ε,0) :
1605 if r_loc = ⊥P then
1606     in⊥S(⊥S)
1607 else if r_loc = ?P then
1608     let info = (past,?S) :
1609     inStreamInfo(info)
1610 else
1611     let (p1,pp1,xp1,pp2,p2) = r_loc :
1612     if xp1 ≠ xp then
1613         in⊥S(⊥S)
1614     else case p1 of
1615         isStep[Position⊥?](pos1) :
1616             let r1 = (pos1,pp1,xp1,pp2,p2) :
1617             inStreamInfo((past,f[xp,sid,r1,c,ts,strinfos_acc]))
1618         isPosition⊥?(pos1) :
1619             if pos1 = ⊥P then
1620                 in⊥S(⊥S)
1621             else if pos1 = ?P then
1622                 let info = (past,?S) :
1623                 inStreamInfo(info)
1624         else
1625             case p2 of
1626                 isStep[Position⊥?](pos2) :
1627                 let r1 = (pos1,pp1,xp1,pp2,pos2) :

```

```

1628         let info = (past, f[xp,sid,r1,c,ts,strinfos_acc])
1629         inStreamInfo(info)
1630     isPosition1?(pos2)      :
1631         if pos2 = 1P then
1632             in1S(1S)
1633         else if pos2 = ?P then
1634             let info = (past,?S) :
1635             inStreamInfo(info) :
1636         else
1637             let streampast = get_stream_past(sid,present) :
1638             if streampast = 1S then
1639                 in1S(1S)
1640             else
1641                 let r1 = (pos1,pp1,xp1,pp2,pos2) :
1642                 let max_pos = maximal_position(streampast) :
1643                 let (from,to) =
1644                     precise_range_boundaries(pos1,pp1,pp2,pos2) :
1645                 if from > max_pos then
1646                     let info = (past, f[xp,sid,r1,c,ts,strinfos_acc])
1647                     inStreamInfo(info)
1648                 else
1649                     if max_pos < to then
1650                         let upper = max_pos :
1651                         let flag = "incomplete" :
1652                     else
1653                         let upper = to :
1654                         let flag = "complete" :
1655                     let new_strinfos_acc =
1656                         update_acc_strinfos[xp,sid,c,ts,strinfos_acc]
1657                         (from,to) :
1658                     case new_strinfos_acc of
1659                         isStep[StreamInfoStar1](step) :
1660                             let info =
1661                                 (past, f[xp,sid,r1,c,ts,strinfos_acc])
1662                             inStreamInfo(info)
1663                         isStreamInfoStar1(new_streaminfos) :
1664                             if new_streaminfos = 1S then
1665                                 in1S(1S)
1666                             else
1667                                 let pst= mergeall(new_streaminfos,ε) :
1668                                 let info =
1669                                     if flag = "complete" then
1670                                         (pst,NIL)
1671                                     else
1672                                         let r2=(upper,pp1,xp,pp2,to):
1673                                         (pst, f[xp,sid,r1,c,ts,
1674                                             new_streaminfos]) :
1675                                         InStreamInfo(info)
1676 : f[XP,SID,Tr[[RAN]],Tr[[CONSTR]],Tr[[TS]],ε]
1677
1678 -----
1679 update_acc_strinfos : XP × SID × Step[ConstrDom] × Stream × StreamInfo*
1680                    → Position × Position
1681                    → Step[StreamInfoStar1]
1682
1683 StreamInfoStar1 = StreamInfoStar* u {1S}
1684 -----
1685
1686 update_acc_strinfos[xp0,sid0,contrait0,ts0,acc_strinfos0](from0,to0) =
1687 let f : XP × SID × Step[ConstrDom] × Stream × StreamInfo*
1688     → Position × Position
1689     → Step[StreamInfoStar1]
1690 f[xp,sid,contrait,ts,acc_strinfos](from,to)(present) =
1691 if from > to then
1692     inAnswer[StreamPast1](acc_strinfos)
1693 else
1694     let constr = contrait(present[xp→from]) :
1695     case constr of
1696         isStep(ConstrDom)(c0) :
1697             inAnswer[StreamPast1]
1698             (update_acc_strinfos[xp,sid,c0,ts,acc_strinfos](from,to))
1699         isConstrDom(c)      :
1700             case c of
1701                 1E          : inAnswer[StreamInfoStar1](1S)

```

```

1702     isPresentMK(pres_mk) :
1703         let pres_mk = (pres,mk) :
1704         if mk = 1F then
1705             inAnswer[StreamInfoStar1](1S)
1706         else if mk ≠ true then
1707             let from1 = from + 1 :
1708             let newacc_strinfos =
1709                 update_acc_strinfos_aux(acc_strinfos,xp,from1,present,ε):
1710             if newacc_strinfos = 1S then
1711                 in1S(1S)
1712             else
1713                 inAnswer[StreamInfoStar1]
1714                     (update_acc_strinfos[xp,sid,c,ts,newacc_strinfos]
1715                      (from1,to))
1716         else
1717             let s1 = is(pres[xp→from]) :
1718             case s1 of
1719                 1S : in1S(1S)
1720                 isStreamInfo(strinfo) :
1721                     let updated_strinfos =
1722                         update_acc_strinfos_aux(acc_strinfos,xp,from,pres,ε):
1723                     if newacc_strinfos = 1S then
1724                         in1S(1S)
1725                     else
1726                         let newacc_strinfos = acc_strinfos||strinfo :
1727                         inAnswer[StreamInfoStar1]
1728                             (update_acc_strinfos[xp,sid,c,ts,newacc_strinfos]
1729                              (from1,to))
1730 : f[xp0,sid0,constraint0,ts0,acc_strinfos0](from0,to0)
1731
1732 -----
1733 update_acc_strinfos_aux :
1734     StreamInfo* × XP × Position × Present × StreamInfo*
1735     → StreamInfoStar1
1736 -----
1737
1738 update_acc_strinfos_aux(strinfos,xp,pos,present,acc) =
1739     if strinfos = ε then
1740         acc
1741     else
1742         let (s,rest) = strinfos :
1743         let (past,future) = s:
1744         case future of
1745             NIL      : let newacc = acc||s
1746             ?S       : let newacc = acc||s
1747             External : let newacc = acc||s
1748             isStream(str) :
1749                 let s1 = str(present[xp→pos])
1750                 case s1 of
1751                     1S : in1S(1S)
1752                     isStreamInfo(strinfo) :
1753                         let (past1,future1) = strinfo :
1754                         let (messages, _) = past :
1755                         let (messages1,l) = past1 :
1756                         let new_messages = messages||messages1 :
1757                         let new_past = (new_messages,l) :
1758                         let new_strinfo = (new_past,future1) :
1759                         let newacc = acc||new_strinfo :
1760                         update_acc_strinfos_aux(rest,xp,pos,newacc)
1761
1762 -----
1763 mergeall : StreamInfo* × StreamPast → StreamPast
1764 -----
1765
1766 mergeall(streaminfos, acc) =
1767     if streaminfos = ε then
1768         acc
1769     else
1770         let (strinfo,rest) = streaminfos :
1771         let (messages,l) = strinfo :
1772         let (messages_acc,_) = acc :
1773         let newmessages = messages_acc||messages :
1774         let newacc = (newmessages,l) :
1775         mergeall(rest,newacc)

```

```

1776
1777 Tr[[construct XP in SID with RAN CONSTR : TV]] =
1778   let f : XP × SID
1779       × Present → Range u {?P,⊥P}
1780       × Step[ConstrDom]
1781       × Step[Value⊥?]
1782   → Stream
1783   f[xp,sid,r,c,tv](present) =
1784     let r_loc = r(present) :
1785     let past = (ε,0) :
1786     if r_loc = ⊥P then
1787       in⊥S(⊥S)
1788     else if r_loc = ?P then
1789       let info = (past,?S) :
1790       inStreamInfo(info)
1791     else
1792       let (p1,pp1,xp1,pp2,p2) = r_loc :
1793       if xp1 ≠ xp then
1794         in⊥S(⊥S)
1795       else case p1 of
1796         isStep[Position⊥?](pos1) :
1797           let r1 = (pos1,pp1,xp1,pp2,p2) :
1798           inStreamInfo((past,f[xp,sid,r1,c,tv]))
1799         isPosition⊥?(pos1)       :
1800           if pos1 = ⊥P then
1801             in⊥S(⊥S)
1802           else if pos1 = ?P then
1803             let info = (past,?S) :
1804             inStreamInfo(info)
1805         else
1806           case p2 of
1807             isStep[Position⊥?](pos2) :
1808               let r1 = (pos1,pp1,xp1,pp2,pos2) :
1809               inStreamInfo(past,f[xp,sid,r1,c,tv]))
1810             isPosition⊥?(pos2)       :
1811               if pos2 = ⊥P then
1812                 in⊥S(⊥S)
1813               else if pos2 = ?P then
1814                 let info = (past,?S) :
1815                 inStreamInfo(info)
1816             else
1817               let streampast = get_stream_past(sid,present) :
1818               if streampast = ⊥S then
1819                 in⊥S(⊥S)
1820             else
1821               let r1 = (pos1,pp1,xp1,pp2,pos2) :
1822               let max_pos = maximal_position(streampast) :
1823               let (from,to) =
1824                 precise_range_boundaries(pos1,pp1,pp2,pos2) :
1825               if from > max_pos then
1826                 let info = (past,f[xp,sid,r1,c,tv]) :
1827                 inStreamInfo(info)
1828             else
1829               if max_pos < to then
1830                 let upper = max_pos :
1831                 let flag = "incomplete" :
1832               else
1833                 let upper = to :
1834                 let flag = "complete" :
1835               let acc = (ε,0),
1836               let strpast =
1837                 construct_past_from_value[xp,sid,c,tv]
1838                 (from,to,acc) :
1839               case strpast of
1840                 isStep[StreamPast⊥](step) :
1841                   let info = (past,f[xp,sid,r1,c,tv]) :
1842                   inStreamInfo(info)
1843                 isStreamPast⊥(pst) :
1844                   if pst = ⊥S then
1845                     in⊥S(⊥S)
1846                   else
1847                     let info =
1848                       if flag = "complete" then
1849                         (pst,NIL) :

```

```

1850                                     else
1851                                         let r2 =
1852                                             (upper, pp1, xp, pp2, to) :
1853                                             (pst, f[xp, sid, r2, c, tv]) :
1854                                             inStreamInfo(info)
1855 : f[XP, SID, Tr[[RAN]], Tr[[CONSTR]], Tr[[TV]]
1856
1857 -----
1858 construct_past_from_value :
1859     XP × SID × Step[ConstrDom] × Step[ValueI?]
1860     → Position × Position × StreamPast
1861     → Step[StreamPastI]
1862
1863 StreamPastI = StreamPastU{IS}
1864 -----
1865
1866 construct_past_from_value[xp0, sid0, constraint0, tv0](from0, to0, acc0) =
1867 f : XP × SID × Step[ConstrDom] × Step[ValueI?]
1868     → Position × Position × StreamPast
1869     → Step[StreamPastI]
1870 f[xp, sid, constraint, tv](from, to, acc)(present) =
1871     if from > to then
1872         inAnswer[StreamPastI](acc)
1873     else
1874         let constr = constraint(present[xp→from]) :
1875         case constr of
1876             isStep(ConstrDom)(c0) :
1877                 inAnswer[StreamPastI](f[xp, sid, c0, tv](from, to, acc))
1878             isConstrDom(c) :
1879                 case c of
1880                     IE : inAnswer[StreamPastI](IS)
1881                     isPresentMK(pres_mk) :
1882                         let pres_mk = (pres, mk) :
1883                         if mk = IF then
1884                             inAnswer[StreamPastI](IS)
1885                         else if mk ≠ true then
1886                             let from1 = from + 1 :
1887                             inAnswer[StreamPastI](f[xp, sid, c, tv](from1, to, acc))
1888                     else
1889                         let v1 = tv(pres[xp→from]) :
1890                         case v1 of
1891                             isStep(ValueI?)(v) :
1892                                 inAnswer[StreamPastI](f[xp, sid, c, tv](from, to, acc))
1893                             isValueI?(v) :
1894                                 if v = IV then
1895                                     inAnswer[StreamPastI](IS)
1896                                 else
1897                                     let current_time = (present↓1)↓1 :
1898                                     let (messages, _) = acc :
1899                                     let newmessages = messages || (current_time, v) :
1900                                     let newacc = (newmessages, 1) :
1901                                     let from1 = from + 1 :
1902                                     inAnswer[StreamPastI](f[xp, sid, c, tv](from1, to, newacc))
1903 : f[[xp0, sid0, constraint0, tv0](from0, to0, acc0)
1904
1905 -----
1906 Tr[[BIND]]: Binder // Present → (Present × Binder) u {IE}
1907 BIND ::= formula XF = F | position XP = TP
1908         | value XV = TV | stream XS = SID
1909 -----
1910
1911 Tr[[formula XF = F]] =
1912     let f : XF × Step[MK] → Binder,
1913     f[xf, f0](present) =
1914         let b0 = f0(present) :
1915         case b0 of
1916             isMK(b) :
1917                 if b = IF then
1918                     IE
1919                 else
1920                     let f1 : Step[MK]
1921                     f1(present) = inAnswer(b0) :
1922                     (present[XF→b0], f[xf, f1])
1923         isStep[MK](s) :

```

```

1924         (present[XF→b0], f[xf,b0])
1925     : f[XF, Tr[[F]]]
1926
1927 Tr[[position XP = TP]] =
1928     let f : XP × Step[Position1?] → Binder,
1929         f[xp,tp0](present) =
1930         let p0 = tp0(present) :
1931         case p0 of
1932             isPosition1?(p) :
1933                 if p = 1P then
1934                     1E
1935                 else
1936                     let f1 : Step[Position1?]
1937                         f1(present) = inAnswer(p) :
1938                         (present[XP→p], f[xf,f1])
1939                     isStep[Position1?](tp1) :
1940                     (present[XF→p0], f[xf,p0])
1941     : f[XP, Tr[[TP]]]
1942
1943 Tr[[value XV = TV]] =
1944     let f : XV × Step[Value1?] → Binder
1945         f[xv,tv0](present) =
1946         let v0 = tv0(present) :
1947         case v0 of
1948             isValue1?(v) :
1949                 if v = 1V then
1950                     1E
1951                 else
1952                     let f1 : Step[Value1?]
1953                         f1(present) = inAnswer(v) :
1954                         (present[xv→v], f[xv,f1])
1955                     isStep[Value1?](tv1) :
1956                     (present[XF→v0], f[xf,v0])
1957     : f[XV, Tr[[TV]]]
1958
1959 Tr[[stream SID = TS]] =
1960     let f : SID × Stream → Binder,
1961         f[sid,stream](present) =
1962         let s0 = stream(present) :
1963         case s0 of
1964             1S : 1E
1965             isStreamInfo(strinfo) :
1966                 let strinfo = (streampast,streamfuture) :
1967                 case streamfuture of
1968                     isNIL :
1969                         let s1 : Stream
1970                             s1(present) = inStreamInfo(strinfo) :
1971                             inBinder(present[sid→s0], f[sid,s1])
1972                     is?S :
1973                         let s1 : Stream
1974                             s1(present) = inStreamInfo(strinfo) :
1975                             inBinder(present[sid→s0], f[sid,s1])
1976                     isExternal :
1977                         let s1 : Stream
1978                             s1(present) = inStreamInfo(strinfo) :
1979                             inBinder(present[sid→s0], f[sid,s1])
1980                     isStream(st) :
1981                         inBinder(present[sid→s0], f[sid,st])
1982     : f[SID, Tr[[TS]]]
1983
1984 -----
1985 Tr[[CONSTR]]: Step[ConstrDom] // ConstrDom = 1E + (Present × MK)
1986 CONSTR ::= ε                               Empty
1987         | satisfying F CONSTR
1988         | BIND CONSTR
1989 -----
1990
1991 Tr[[ε]](present) =
1992     let pres_tr = (present,true) :
1993     inAnswer[ConstrDom](pres_tr)
1994
1995 Tr[[satisfying F CONSTR]] =
1996     let f : Step[MK] × Step[ConstrDom] → Step[ConstrDom]

```

```

1997     f[f0,c0](present) =
1998         let b0 = f0(present) :
1999         case b0 of
2000             isMK(b) :
2001                 if b ≠ true then
2002                     let ans = (present,b) :
2003                     inAnswer[ConstrDom](ans)
2004                 else
2005                     let ans = c0(present) :
2006                     inAnswer[ConstrDom](ans)
2007             isStep[MK](f1) :
2008                 inAnswer[ConstrDom](f[f1,c0])
2009 : f[Tr[[F]],Tr[[CONSTR]]]
2010
2011 Tr[[BIND CONSTR]](present) =
2012     let f : Binder × Step[ConstrDom] → Step[ConstrDom]
2013     f[bind0,c0](present) =
2014         let b0 = bind0(present) :
2015         if e0 = ⊥E then
2016             ⊥E
2017         else
2018             let b0 = (present1,binder1) :
2019             let ans = c0(present1) :
2020             inAnswer[ConstrDom](ans)
2021 : f[Tr[[BIND]], Tr[[CONSTR]]]
2022
2023 -----
2024 Tr[[RAN]]: Present → Range u {?P,⊥P}
2025 RAN ::= TP1 PP1 XP PP2 TP2 | TP PP XP
2026 -----
2027
2028 // Type checker should guarantee that TP1 and TP2 are positions in the
2029 // same stream XP refers to.
2030
2031 Tr[[TP1 PP1 XP PP2 TP2]] =
2032     let r : Range → Present → Range u {?P,⊥P}
2033     r(tp1,pp1,xp,pp2,tp2)(present) =
2034         let p1 = tp1(present) :
2035         case p1 of
2036             is?P : ?P
2037             is⊥P : ⊥P
2038             isPosition(pos1) :
2039                 let p2 = tp2(present) :
2040                 case p2 of
2041                     is?P : ?P
2042                     is⊥P : ⊥P
2043                     isPosition(pos2) : (pos1,PP1,xp,PP2,pos2)
2044                     isStep[Position⊥? u {∞}](s2) : (pos1,PP1,xp,PP2,s2)
2045             isStep[Position](s1) :
2046                 let p2 = tp2(present) :
2047                 case p2 of
2048                     is?P : ?P
2049                     is⊥P : ⊥P
2050                     isPosition(p2) : (s1,PP1,xp,PP2,pos2)
2051                     isStep[Position⊥? u {∞}](s2) : (s1,PP1,xp,PP2,s2)
2052
2053     : r(Tr[[TP1]],PP1,XP,PP2,Tr[[TP2]])
2054
2055 Tr[[TP PP XP]] =
2056     let r : Range → Present → Range u {?P,⊥P}
2057     r(tp1,pp1,xp,pp2,tp2) =
2058         let p = tp1(present) :
2059         case p of
2060             is?P : ?P
2061             is⊥P : ⊥P
2062             isPosition(pos) : (pos,pp1,xp,pp2,tp2)
2063             isStep[Position](s) : (s,pp1,xp,pp2,tp2)
2064     : r(Tr[[TP]],PP,XP,"<",∞)
2065
2066 -----
2067 Tr[[PV]] : Present → VPSn → MK
2068 -----
2069

```

2070 Tr[[PV]](message,stream\_id,envs,envf,envp,envv,envfunpr) = envfunpr(PV)

2071

2072 -----

2073 Tr[[FV]] : Present → VPS<sup>n</sup> → Value<sub>1</sub>?

2074 -----

2075

2076 Tr[[FV]](message,stream\_id,envs,envf,envp,envv,envfunpr) = envfunpr(FV)

2077

2078 -----

2079 Tr[[FS]] : Present → VPS<sup>n</sup> → Stream

2080 -----

2081

2082 Tr[[FS]](message,stream\_id,envs,envf,envp,envv,envfunpr) = envfunpr(FS)