Analytic and Algorithmic Methods for Computing the Space Requirements of Stream Monitor Specifications

David M. Cerna

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Temur Kutsia

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Abstract

In previous work "Better Space Bounds for Future-Looking Stream Monitors", upper bounds were derived for the space requirements of monitoring steam specifications in the LogicGuard specification language. The analysis was based on the assumption that in these formulas quantifiers can be arbitrarily nested, but in the body of each quantifier there is at most one quantifier; we refer to such formulas as *quantifier vectors*. While analytic results for two special classes of quantifier vectors were derived, bounds for the space requirements of arbitrary quantifier vectors remained unresolved. In this work, we show how to generalize our previous results in order to derive bounds for any quantifier vector. Based on this generalization, we give an algorithm which can compute space bounds for the full specification language.

Key words: Space complexity, Predicate logic, Runtime verification

⋆ LogicGuard II: The Optimized Checking of Time-Quantified Logic Formulas with Applications in Computer Security is sponsored by the FFG BRIDGE program, project No. 846003.

Email addresses: David.Cerna@risc.jku.at (David M. Cerna), ,Wolfgang.Schreiner@risc.jku.at (Wolfgang Schreiner), Temur.Kutsia@risc.jku.at (Temur Kutsia).

URLs: https://www.risc.jku.at/home/dcerna (David M. Cerna),

1. Introduction

The field of runtime verification is focused on the investigation of methods for checking that formally specified security or safety properties are obeyed, at runtime, during the execution of a system. The specification is translated to an operational form which observes the system and reports violations. For this purpose, complex system executions are abstracted, and the runtime state and history of the system are represented as a stream of events (potentially infinite). It is the stream of events which is monitored by the operational form of the specification. In this paper we generalize the results of"Better Space Bounds for Future-Looking Stream Monitors"(Cerna et al., 2016a) from special cases arbitrary specifications. As before, we measure efficiency of a specification in terms of the memory used during execution. This metric is referred to as the *runtime representation size*.

Both this work and (Cerna et al., 2016a) investigate the space complexity of the LogicGuard stream monitor specification language (LogicGuard II, 2015; Schreiner et al., 2015), which was developed in an industrial collaboration for the runtime monitoring of networks for security violations. The driving force behind the development of LogicGuard was to use an alternative framework to the language of linear temporal logic (LTL) (Maler et al., 2005; Armoni et al., 2002; Banieqbal and Barringer, 1987; Rosu and Bensalem, 2006). LTL can be used for constructing efficient stream monitors, but due to its limited expressive power, complex properties, of interest to us and our industrial partners, are difficult to express. The LogicGuard specification language was developed from a monadic predicate logic foundation with the addition of limited set construction, arithmetic, and some computational power (fold operations). This, of course, results in a language more expressive than LTL because it is based on a larger fragment of predicate logic. These extra concepts allow the construction of new event streams from the external streams that belong to the system being monitored, which allows us to raise the level of abstraction even further.

Though allowing these additional concepts in our specifications provides powerful abstraction and flexibility, it also costs a lot in terms of efficiency. Many of the possible specifications are too expensive to monitor effectively. Thus, an important goal since the seminal work of the LogicGuard language development has been finding efficient ways to describe properties of interest in this language. The first step towards this goal was to perform a static analysis in order to determine whether a specification gives rise to a monitor operating with only a finite set of past messages. This static analysis was shown to be sound (Kutsia and Schreiner, 2014) and in the end resulted in a "history pruning" optimization. The full specification language is quite complex, but many of its core features can be described in a much simpler language, the core language. The proof of soundness for this optimization was carried out using the core language. In this work, we will present the majority of the results in this simpler language, however, the algorithm introduced in Section 7 works for the full specification language after a translation is performed. The results of (Cerna et al., 2016a) could not be applied to the full specification language without syntactic restrictions.

The history analysis only considers one facet of the system which can help with defining efficient specifications. In (Cerna, 2015b; Cerna et al., 2016c), a complementary analysis

---

`https://www.risc.jku.at/home/schreine` (Wolfgang Schreiner),
`https://www.risc.jku.at/home/tkutsia` (Temur Kutsia).

was carried out to determine the space requirements of the "future" looking parts of the operational form of the monitors. In (Cerna et al., 2016a), we focused on two special cases. A method was developed to analyse the number of quantified formula preserved in memory during runtime. They are kept in memory because their truth values cannot be determined from current observations. Together with the history analysis, we get an idea of the runtime representation size of a monitor as well as the time required for processing the monitor's operational form. In (Cerna, 2015b), we were able to distinguish between specifications requiring infinite memory and those requiring finite memory, similar to (Kutsia and Schreiner, 2014) which distinguished finite from infinite history. In (Cerna et al., 2016c), the focus was on the finite case and an upper bound was derived. In (Cerna et al., 2016a), we went one step further and attempted to derive better bounds for the finite case. In the end, we where able to construct much better bounds, but at the same time, these bounds were only applicable to very specific formula structures. It was conjectured that bounds provided by these formulas represent the bounds for the runtime representation size of any permutation of the quantifiers used in the formula. Permutations of the quantifiers is a sensible concept in our analysis, because we do not consider the propositional formula structure for computing space complexity. Though, these bounds are not as precise as the specific cases they would be much more precise than the results of (Cerna et al., 2016c). We address this issue in Sec. 4 of this paper.

Concerning the formulas we are analysing, the core language of the LogicGuard framework has much in common with Monadic First-Order Logic (MFO), see (McNaughton and Papert, 1971). It is well known that LTL (Schnoebelen, 2003b) formulas are translatable into MFO formulas, which capture the class of star-free languages. The full language, on the other hand is closely related to Monadic Second-Order Logic (MSO) (Büchi, 1960), which captures the class of omega-regular languages, and is thus more expressive than LTL. Most space complexity results with respect to MFO and MSO use as a measure the size of the non-deterministic Büchi automaton that accepts the language of a formula. For MFO, the automaton size is in the worst case exponential with respect to the formula size (Vardi and Wolper, 1986). When a runtime verification method relies on automata-based model checking where the automaton is indeed constructed, this complexity result is applicable. Moreover, the previous result considers non-deterministic automata and in practice determinisation must be performed resulting in a second exponential blow-up. MSO fairs worse in that the size of the accepting automaton is in general non-elementary with respect to the formula size (Frick and Grohe, 2004). This makes the use of MSO as a specification language seem quite impractical.

Even in the case of MFO, a doubly exponential sized automaton is still impractical for runtime verification, forcing subclasses of these logics are considered. The hardware design language PSL (IEEE, 2007) which is based on LTL defines a "simple subclass" that restricts the general use of disjunction in a specification to avoid exponential blowup. In (Kupferman et al., 2006), the class of "locally checkable" properties (a subclass of the "locally testable" properties introduced in (McNaughton and Papert, 1971) is defined, where a word satisfies a property, if every k-length subword of the word (for some $k \in \mathbb{N}$) satisfies this property; such properties can be recognized by deterministic automata whose number of states is exponential in k but independent of the formula size. In (Finkbeiner and Kuhtz, 2009) a procedure for synthesizing monitor circuits from LTL specifications is defined that restricts the exponential blow-up to those parts of a formula that involve unbounded-future operators.

Another related set of works, but only in a dual sense, is the study of the computational complexity of *Interval Temporal Logic* (ITL) (Allen, 1983; Halpern and Shoham, 1991). Most of the analysis of ITL is performed with a focus on the difficulty of model checking formulas and the size of the Kripke structures corresponding to the given fragment of ITL, thus again, complexity is defined by automata size. However, for some severely restricted classes, space complexity results have been devised in terms of the smallest satisfying path in the Kripke Structure (Bozzelli et al., 2016), as well as results concerning the overall memory use for model checking certain fragments (Molinari et al., 2015). For some of these restricted classes it has been shown that there is a relatively small, hard kernel (the size of which is logarithmic in terms of input size) which makes the model checking procedure NP-hard (Gottlob, 1995; Schnoebelen, 2003a). We do not delve into the relationship of these results to our specification in this paper, but we do borrow some of the assumptions from these restricted classes. For example, there is an interval temporal modality which enforces a specific starting time; this modality is integral in separating the hard classes from the simple ones. In this paper, all quantifiers will have the same lower bound, as least when the focus of our investigation is bounds for evaluation vectors, which greatly simplifies the analysis and has led to very useful results.

Unlike these investigations, we do not consider the translation of formulas to automata and do not use automata sizes as the space complexity measure. Rather we base our investigations directly on an operational semantics of formula evaluation which exhibits the formula instances that are kept in memory during each evaluation step. In (Cerna, 2015a; Cerna et al., 2016a), we abstracted the operational semantics of the core language in order to focus specifically on the rules which deal with future looking quantifiers. We also introduced the concepts of *quantifier vectors* and *evaluation vectors* to represent precisely these parts of the monitor which require future positions of the stream. Using these two abstractions we were able to get optimal space complexity results (analytic expressions) for the three important classes of monitors represented by *uniform standard evaluation vectors*, *standard evaluation vectors*, and *inverse standard evaluation vectors*. In this work, we first repeat these results as well as generalize them to get space bounds for general evaluation vectors, which are an abstract representation of any purely future looking monitor. Further analysis of these results allows us to weaken some of the restrictions on evaluation vectors and still derive bounds, however at the loss of providing analytic expressions.

A ubiquitous technique used throughout our work is converting a monitor specification into a *dominating monitor* specification. A deeper analysis of the work presented in (Cerna, 2015a; Cerna et al., 2016a) and the extensions presented here resulted in an algorithmic procedure for computing the size of the runtime representation as precisely as the analytic expressions did for the restricted classes (Cerna et al., 2016b). This algorithm is precise for the class of *dominating monitors*. Though time complexity-wise, the algorithm is less efficient, this does not matter much given that we only need to compute the runtime representation once prior to execution. This algorithm can be used to bound the runtime representation size of any monitor specification written using the core language. To transition to bounding monitor specifications of the full language we wrote a translation function and implemented it in the LogicGuard system (LogicGuard II, 2015). No theoretic results are presented concerning this translation. The runtime representation size and translation can be found in the newest LogicGuard release (LogicGuard II, 2015).

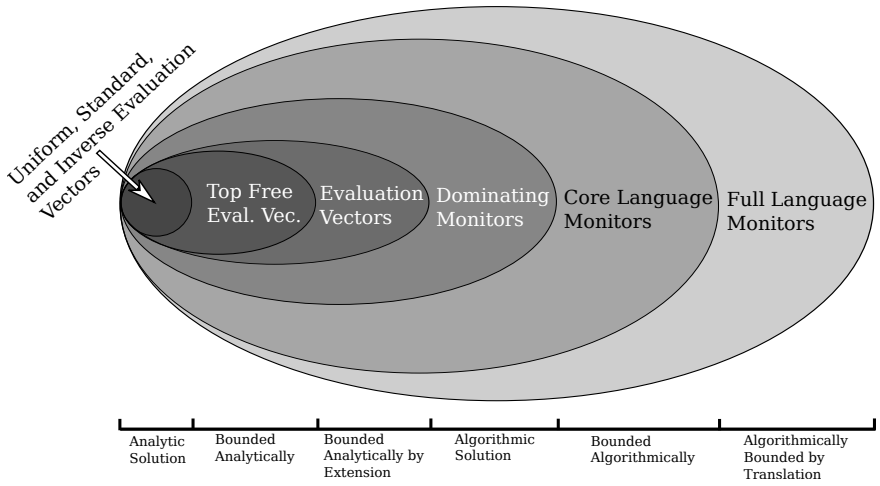

Fig. 1. Hierarchy of results presented in this paper.

Fig. 1 describes the relationship between the various results presented in this paper. The left most classes are the most restricted monitor specifications, and thus we have derived the most precise results for these classes, analytic expressions. As we move to the right of Fig. 1, the classes get more expressive and our derived bounds get less precise. For example, we are able to derive bounds for evaluation vectors by computing a "completion" of the given vector which makes it *top-free*. The size of the runtime representation of the "completion" is always larger than the original vector. The bounds for the full language are the least precise and are the result of a very loose translation to the core language. This translation results in an over-approximated (but reasonable) bound for the runtime representation size.

We provide proofs (partially cited, partially sketches) for most of the propositions in this paper; the major exception is Thm. 44 which links the operational semantics of the core specification language to the abstract framework used for analysis in the results of this paper. An informal proof was provided in (Cerna et al., 2016a) describing the relationship, but a detailed proof is very tedious and from the experience of the authors, such a proof would not be as illuminating as the amount of time required would suggest it to be.

We also would like to make note of the connection between this work and previous publications of the authors. The formalism presented in this paper is almost identical to the formalism of (Cerna et al., 2016a), though we make minor improvements. We also repeat results found in (Cerna et al., 2016a), though without proof. These results are important for the generalizations we present in this work. That is, using the framework of (Cerna et al., 2016a), we complete the analysis of classes discussed but not analysed in (Cerna et al., 2016a) and prove important relationships not discussed in the previous work.

The rest of this paper is structured as follows: in Sec. 2, we present the core of the LogicGuard specification language and its operational semantics. In Sec. 3, we introduce the concept of evaluation vectors, as well as an abstraction of the operational semantics to evaluate them. In Sec. 4, we provide an optimal analytic solution for the space requirements of uniform, standard, and inverse-standard evaluation vectors. In Sec. 5, we

provide bounds for sets of *top-free evaluation vectors* constructed from a set of uniform standard evaluation vectors. In Sec. 6, we provide bounds for general evaluation vectors. In Sec. 7, we describe an algorithm which is precise for dominating monitors and bounds monitor specifications of the core language. In Sec. 8, we conclude by judging the presented analysis and outlining a few open problems which we would like to address in future work.

2. Core Language

We provide a short introduction to the LogicGuard Specification language in order to aid understanding of the core language and its operation semantics; the full language is described in (Schreiner et al., 2015).

2.1. LogicGuard Specification Language

The LogicGuard language for monitoring event streams allows the derivation of higher level streams (representing e.g. a sequence of messages transmitted by the datagrams) from a lower level input stream (representing e.g. a sequence of TCP/IP datagrams). These higher level streams are processed by a monitor which asserts that a particular property holds (e.g. that every message is within a certain time bound followed by another message whose value is related in a particular way to the value of the first one). These specifications commonly have the following form:

```
type tcp; type message; ...
stream<tcp> IP;
stream<message> S = stream<IP> x satisfying start(@x) :
  value[seq,@x,combine]<IP> y
      with x < _ satisfying same(@x,@y) until end(@y) :
    @y ;
monitor<S> M = monitor<S> x satisfying trigger(@x) :
  exists<S> y with x < _ <=# x+T:
    match(@x,@y);
```

After the declaration of types `tcp` and `message` and external functions and predicates operating on objects of these types, a stream `IP` of TCP/IP datagrams is declared that is connected by the runtime system to the network interface. From this stream, a "virtual" stream `S` of "messages" is derived; each message is created by sequentially combining every datagram at position `x` on `IP` (whose value is denoted by `@x`) that satisfies a predicate `start` by application of a function `combine` with every subsequent datagram at position `y` that is related to the first one by a predicate `same` until a termination condition `end` is satisfied. For this purpose we provide the construct `value` [`seq`, b, f] x `with` c t which computes the value $f(f(\cdots f(b, t_1), t_2) \cdots, t_n)$, where $t_1, \cdots t_n$ are those values of t when the variable x is assigned all values determined by constraint c. The stream `S` is monitored by a monitor `M` that checks whether for every message on `S` that satisfies a `trigger` predicate within `T` time, a partner message appears that fits with the first message according to some `match` predicate.

To support a formal analysis, in (Kutsia and Schreiner, 2014) a core version of the LogicGuard language (see Fig. 2) was defined and given a formal operational semantics. This core language has been subsequently used to analyse the complexity of monitoring

$$
\begin{array}{lll}
M ::= & \forall_{0\leq V} : F. & m ::= \quad \forall^{\mathbb{P}(\mathbb{N}\times f\times c)}_{0\leq V} : f \\
F ::= & @V \mid \neg F \mid F \ \wedge \ F \mid F \ \& \ F & f ::= \quad \mathbf{d}(\top) \mid \mathbf{d}(\bot) \mid \mathbf{n}(g) \\
\mid & \forall_{V\in[B,B]} : F. & g ::= \quad @V \mid \neg f \mid f \ \wedge \ f \mid f \ \& \ f \\
B ::= & 0 \mid \infty \mid V \mid B \pm N. & \mid \quad \forall_{V\in[b,b]} : f \mid \forall_{V\in[\mathbb{N}^\infty,\mathbb{N}^\infty]} : f \\
V ::= & x \mid y \mid z \mid \ \ldots & \mid \quad \forall^{\mathbb{P}(\mathbb{N}\times f\times c)}_{V\leq\mathbb{N}^\infty} : f \\
N ::= & 0 \mid 1 \mid 2 \mid \ \ldots & b ::= \quad c \to \mathbb{N}^\infty \\
 & & c ::= \quad (V \to^{\text{part.}} \mathbb{N}) \times (V \to^{\text{part.}} \{\top, \bot\})
\end{array}
$$

$$
\begin{array}{rl}
T(\forall_{0\leq V} : F) & := \forall^{\emptyset}_{0\leq V} : T^{\mathrm{F}}(F) \\
T^{\mathrm{F}}(@V) & := \mathbf{n}(@V) \\
T^{\mathrm{F}}(\neg F) & := \mathbf{n}(\neg T^{\mathrm{F}}(F)) \\
T^{\mathrm{F}}(F_1 \ \wedge \ F_2) & := \mathbf{n}(T^{\mathrm{F}}(F_1) \wedge T^{\mathrm{F}}(F_2)) \\
T^{\mathrm{F}}(F_1 \ \& \ F_2) & := \mathbf{n}(T^{\mathrm{F}}(F_1) \ \& \ T^{\mathrm{F}}(F_2)) \\
T^{\mathrm{F}}(\forall_{V\in[B_1,B_2]} : F) & := \forall_{V\in[T^B(B_1),T^B(B_2)]} : T^{\mathrm{F}}(F)
\end{array}
$$

$$
\begin{array}{rl}
T^{\mathrm{B}}(0) & := \lambda c.\ 0 \\
T^{\mathrm{B}}(\infty) & := \lambda c.\ \infty \\
T^{\mathrm{B}}(V) & := \lambda c.\ c.1(V) \\
T^{\mathrm{B}}(B \pm N) & := \lambda c.\ T^{\mathrm{B}}(B)(c) \pm N
\end{array}
$$

Fig. 2. The core language: syntax, runtime representation, translation.

and to derive the results presented in this paper. The analysis was also implemented in the LogicGuard system by translating specifications from the full language to the core language such that the analysis of the translated specification also predicts the complexity of monitoring the original specification (the translation is not semantics-preserving but generates a specification for which monitoring is at least as complex as the monitoring of the initial specification).

The syntax of this core language is depicted on the lefthand side of Fig. 2 where the typed variables $M, F, \ldots$ denote elements of the syntactic domains $\mathbb{M}, \mathbb{F}, \ldots$ of monitors, formulas, etc. A monitor M has the form $\forall_{0\leq V} : F$ for some variable V and formula F; it processes an infinite stream of truth values $\top$ (true) or $\bot$ (false) by evaluating F for $V = 0, V = 1, \ldots$ The predicate $@V$ denotes the value in the stream at position V, $\neg F$ denotes the negation of F, $F_1 \ \wedge \ F_2$ denotes parallel conjunction (both F_1 and F_2 are evaluated simultaneously), $F_1 \ \& \ F_2$ denotes sequential conjunction (the evaluation of F_2 is delayed until the value of F_1 becomes available), $\forall_{V\in[B_1,B_2]} : F$ denotes universal quantification over the interval $[B_1, B_2]$.

A monitor $M \in \mathbb{M}$ is translated by the function $T : \mathbb{M} \to \mathcal{M}$ defined at the bottom of Fig. 2 into its runtime representation $m = T(M) \in \mathcal{M}$ whose structure is depicted on the right-hand side; here the typed variables $m, f, \ldots$ denote elements of the runtime domains $\mathcal{M}, \mathcal{F}, \ldots$, i.e., the runtime representations of $M, F, \ldots$ Over the domain $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ arithmetic operations are interpreted in the usual way, i.e., the operator $-$ is interpreted

as truncated subtraction and for every $n \in \mathbb{N}$ we have $\infty \pm n = \infty$ (we assume $0 \in \mathbb{N}$). The notions $\mathbb{P}(S)$ and $A \to^{\text{part.}} B$ denote the powerset of S and the set of partial mappings from A to B, respectively. A context c consists of a pair of partial functions that assign to every variable its position and the truth value that the stream holds at that position, respectively. Though mapping variable to the the position seems redundant, this mapping is necessary for soundness of the history pruning optimization introduced in (Kutsia and Schreiner, 2014).

During the execution of a monitor M, the set I in its runtime representation $T(M) = \forall^I_{0 \le V} : f$ contains those instances of its body F which could not yet be evaluated to $\top$ or $\bot$; each such instance is represented by a tuple (p, f, c) where p is the position assigned to V, f is the (current) runtime representation of F, and c represents the context to be used for the evaluation of f. A runtime representation f can be a tagged value $\mathbf{n}(g)$ where g represents the runtime representation of the formula to be evaluated in the **n**ext step; when the evaluation has completed, its value becomes $\mathbf{d}(t)$ when the evaluation is **d**one, the truth value t represents the result.

2.2. Operational Semantics

The evaluation of a monitor's runtime representation is formally defined by a small-step operational semantics with a 6-ary transition relation

$$\to \subseteq \mathcal{M} \times \mathbb{N} \times \{\top, \bot\}^\omega \times \{\top, \bot\} \times \mathbb{P}(\mathbb{N}) \times \mathcal{M}.$$

A transition is of the form $m \to_{p,s,v,R} m'$ where m is the runtime representation of the monitor prior to the transition, m' is its representation after the transition, p is the stream position of the next message value v to be processed, s denotes the sequence of p messages that have previously been processed, and R denotes the set of those positions which are reported by the transition relation to make the monitor body false. The monitor thus processes a stream $\langle v_0, v_1, \ldots \rangle$ by a sequence of transitions

$$\left(\forall^{I_0}_{0 \le x} : f\right) \to_{0,s_0,v_0,R_0} \left(\forall^{I_1}_{0 \le x} : f\right) \to_{1,s_1,v_1,R_1} \left(\forall^{I_2}_{0 \le x} : f\right) \to \cdots$$

where $s_p = \langle v_0, \ldots, v_{p-1} \rangle$. Each set I_p contains those instances of the monitor which, by the p messages processed so far, could not be evaluated to a truth value yet and each set R_p contains the positions of those instances that were reported to become false by transition p. In particular, we have

$$I_{p+1} = \{(t, \mathbf{n}(g), c) \in \mathcal{I} \mid \exists f \in \mathcal{F} : (t, f, c) \in I' \wedge \vdash f \to_{p,s_p,v_p,c} \mathbf{n}(g)\}$$
$$R_{p+1} = \{t \in \mathbb{N} \mid \exists f \in \mathcal{F}, c \in \mathcal{C} : (t, f, c) \in I' \wedge \vdash f \to_{p,s_p,v_p,c} \mathbf{d}(\bot)\}$$

where $I' = I_p \cup \{(p, f, ((V, p), (V, v_p)))\}$. The transition relation on monitors depends on a corresponding transition relation $f \to_{p,s,v,c} f'$ on formulas where c represents the context for the evaluation of f. In each step p of the monitor transition, a new instance of the monitor body F is added to set I_p, and all instances in that set are evaluated according to the formula transition relation. Note that each formula instance in that set contains the runtime representation of a quantified formula (otherwise, it could have been immediately evaluated) which in turn contains its own instance set; thus instance sets are nested up to a depth that corresponds to the quantification depth of the monitor.

Atomic Formulas		
#	Transition	Constraints
A1	$\mathbf{n}(@y) \rightarrow \mathbf{d}(c.2(y)))$	$y \in dom(c.2)$
A2	$\mathbf{n}(@y) \rightarrow \mathbf{d}(\bot)$	$y \notin dom(c.2)$
Negation		
N1	$\mathbf{n}(\neg f) \rightarrow \mathbf{n}(\neg \mathbf{n}(f'))$	$f \rightarrow \mathbf{n}(f')$
N2	$\mathbf{n}(\neg f) \rightarrow \mathbf{d}(\bot)$	$f \rightarrow \mathbf{d}(\top)$
N3	$\mathbf{n}(\neg f) \rightarrow \mathbf{d}(\top)$	$f \rightarrow \mathbf{d}(\bot)$
Sequential conjunction		
C1	$\mathbf{n}(f_1 \ \& \ f_2) \rightarrow \mathbf{n}(\mathbf{n}(f_1') \ \& \ f_2)$	$f_1 \rightarrow \mathbf{n}(f_1')$
C2	$\mathbf{n}(f_1 \ \& \ f_2) \rightarrow \mathbf{d}(\bot)$	$f_1 \rightarrow \mathbf{d}(\bot)$
C3	$\mathbf{n}(f_1 \ \& \ f_2) \rightarrow \mathbf{n}(f_2')$	$f_1 \rightarrow \mathbf{d}(\top)$, $f_2 \rightarrow \mathbf{n}(f_2')$
Parallel conjunction		
P1	$\mathbf{n}(f_1 \wedge f_2) \rightarrow \mathbf{n}(\mathbf{n}(f_1') \wedge \mathbf{n}(f_2'))$	$f_1 \rightarrow \mathbf{n}(f_1'),\ \ f_2 \rightarrow \mathbf{n}(f_2')$
P2	$\mathbf{n}(f_1 \wedge f_2) \rightarrow \mathbf{n}(f_1')$	$f_1 \rightarrow \mathbf{n}(f_1'),\ \ f_2 \rightarrow \mathbf{d}(\top)$
P3	$\mathbf{n}(f_1 \wedge f_2) \rightarrow \mathbf{n}(f_2')$	$f_2 \rightarrow \mathbf{n}(f_2'),\ \ f_1 \rightarrow \mathbf{d}(\top)$
P4	$\mathbf{n}(f_1 \wedge f_2) \rightarrow \mathbf{d}(\bot)$	$f_2 \rightarrow \mathbf{n}(f_2'),\ \ f_1 \rightarrow \mathbf{d}(\bot)$
P5	$\mathbf{n}(f_1 \wedge f_2) \rightarrow \mathbf{d}(\bot)$	$f_1 \rightarrow \mathbf{n}(f_1'),\ \ f_2 \rightarrow \mathbf{d}(\bot)$
Quantification		
Q1	$\forall_{y \in [b_1, b_2]} : f \rightarrow \mathbf{d}(\top)$	$p_1 = b_1(c)$, $p_2 = b_2(c)$, $p_1 > p_2 \vee p_1 = \infty$
Q2	$\forall_{y \in [b_1, b_2]} : f \rightarrow F'$	$p_1 = b_1(c)$, $p_2 = b_2(c), p_1 \neq \infty$, $p_1 \leq p_2$, $\mathbf{n}(\forall_{y \in [p_1, p_2]} : f) \rightarrow F'$
Q3	$\mathbf{n}(\forall_{y \in [p_1, p_2]} : f) \rightarrow \mathbf{n}(\forall_{y \in [p_1, p_2]} : f)$	$p < p_1$
Q4	$\mathbf{n}(\forall_{y \in [p_1, p_2]} : f) \rightarrow F'$	$p_1 \leq p$, $\mathbf{n}(\forall^{I_0}_{y \leq p_2} : f) \rightarrow F'$
Q5	$\mathbf{n}(\forall^{I}_{y \leq p_2} : f) \rightarrow \mathbf{d}(\bot)$	DF
Q6	$\mathbf{n}(\forall^{I}_{y \leq p_2} : f) \rightarrow \mathbf{d}(\top)$	$\neg DF, I'' = \emptyset, p_2 < p$
Q7	$\mathbf{n}(\forall^{I}_{y \leq p_2} : f) \rightarrow \mathbf{n}(\forall^{I''}_{y \leq p_2} : f)$	$\neg DF, (I'' \neq \emptyset \vee p \leq p_2)$

Fig. 3. The operational semantics of formula evaluation.

Figure 3 shows an excerpt of the operational semantics of formula evaluation (a more thorough discussion can be found in (Kutsia and Schreiner, 2014)) where the transition arrow $\rightarrow$ is to be read as $\rightarrow_{p,s,v,c}$ and rules Q4–Q7 are based on the following definitions:

$$I_0 = \{(i, f, (c.1[V \mapsto i], c.2[V \mapsto s(i + p - |s|)])) \mid p_1 \leq i \leq \min\{p_2 + 1, p\}\}$$

$$I' = \begin{cases} I & \text{if } p_2 < p \\ I \cup (p, f, (c.1[V \mapsto p], c.2[V \mapsto v])) & \text{otherwise} \end{cases}$$

$$I'' = \{(t, \mathbf{n}(g), c) \in \mathcal{I}' \mid (t, f, c) \in I' \wedge \vdash f \rightarrow \mathbf{n}(g)\}$$

$$DF \equiv \exists t \in \mathbb{N}, f \in \mathcal{F}, c \in \mathcal{C} : (t, f, c) \in I' \wedge \vdash f \rightarrow \mathbf{d}(\bot)$$

Essentially, rules Q1, Q2, and Q3 deal with quantifiers of which are either malformed, have an uninstantiated stream variable, or the lower end of the interval is greater than the current stream position. If the quantifier is correctly formed and the lower end of the interval is greater than or equal the current stream position, we can move on to the application of Q4 which adds an instance set to the runtime representation of the quantifier. The rules Q5 and Q6 check if any instance from the instance set evaluated to false or if every instance in the instance set evaluated to true and the stream position is beyond the upper bound of the quantifier. When neither Q5 or Q6 can be applied we apply Q7 upon transition to the next position in the stream. New instances are added to the instance set of the quantifier. We provide an example adapted from (Cerna et al., 2016c) concerning the application of these rules.

Example 1. Consider the monitor $M = \forall_{0 \leq x} : \forall_{y \in [x+1, x+2]} : @x \ \& \ @y$, which states that the current position of the stream is true as well as the next two future positions. We determine its runtime representation $m = T(M)$ as $m = \forall^{\emptyset}_{0 \leq x} : f$ with $f = \forall_{y \in [b_1, b_2]} : g$ for some b_1 and b_2 and $g = @x \ \& \ @y$. We evaluate m over the stream $\langle \top, \top, \bot, \ldots \rangle$. First consider the transition $(\forall^{\emptyset}_{0 \leq x} : f) \rightarrow_{0, \langle\rangle, \top, \emptyset} (\forall^{I_0}_{0 \leq x} : f)$. which generates the instance set

$$I^0 = \left\{(0, \mathbf{n}(\forall_{y \in [1,2]} : g), (\{(x, 0)\}, \{(x, \top)\}))\right\}.$$

Performing another step $(\forall^{I_0}_{0 \leq x} : f) \rightarrow_{1, \langle\top\rangle, \top, \emptyset} (\forall^{I_1}_{0 \leq x} : f)$ we get

$$I^1 = \begin{array}{l} \big\{(1, \mathbf{n}(\forall_{y \in [2,3]} : g), (\{(x, 1)\}, \{(x, \top)\})), \\ \quad (0, \mathbf{n}(\forall^{\emptyset}_{y \leq 2} : g), (\{(x, 0)\}, \{(x, \top)\}))\big\} \end{array}.$$

The instance set $\emptyset$ in the runtime representation of the formula is empty, because the body of the quantified formula is propositional and evaluates instantly. Notice that the new instance is the same as the instance in I^0 but the positions are shifted by 1. The next step is $(\forall^{I_1}_{0 \leq x} : f) \rightarrow_{2, \langle\top, \top\rangle, \bot, \{0,1\}} (\forall^{I_2}_{0 \leq x} : f)$ where

$$I^2 = \left\{(2, \mathbf{n}(\forall_{y \in [3,4]} : g), (\{(x, 2)\}, \{(x, \bot)\}))\right\}.$$

The first two instances evaluate at this point and both violate the specification, thus yielding the set $\{0, 1\}$ of violating positions of the monitor. Again, the remaining instance is shifted by one position.

Our goal is to determine for arbitrary sequences

$$\left(\forall^{I_0}_{0 \leq x} : f\right) \rightarrow_{0, s_0, v_0, R_0} \left(\forall^{I_1}_{0 \leq x} : f\right) \rightarrow_{1, s_1, v_1, R_1} \left(\forall^{I_2}_{0 \leq x} : f\right) \rightarrow \cdots$$

arising for some monitor $M = \forall_{0\leq x} : f$ the maximum size of I_p plus, for each $q = \mathbf{n}(\forall^I_{y\leq p_2} : f) \in IS^{f_i}$, the size of I. We formalize this problem in the following subsection.

2.3. Runtime Representation Size

Now we address the central problem of this work. Our goal is to determine the maximum size of the runtime representation of a monitor during its execution. For doing this we have to define the size of the runtime representation of monitors, formulas and formula instances.

Definition 1. We define the functions $c_\mathrm{m} : \mathcal{M} \to \mathbb{N}$, $c_\mathrm{f} : \mathcal{F} \to^{\text{part.}} \mathbb{N}$, $c_\mathrm{g} : \mathcal{G} \to \mathbb{N}$, and $c_\mathrm{i} : \mathcal{I} \to \mathbb{N}$ which denote the size of the runtime representation of a monitor respectively unevaluated formula (with and without tag) respectively formula instance:

$$\begin{aligned}
c_\mathrm{m}(\forall^I_{0\leq V} : f) &= \textstyle\sum_{g\in I} c_\mathrm{i}(g) & c_\mathrm{f}(\mathbf{n}(g)) &= c_\mathrm{g}(g)\\
c_\mathrm{g}(@V) &= 0 & c_\mathrm{g}(f_1 \ \wedge \ f_2) &= c_\mathrm{f}(f_1) + c_\mathrm{f}(f_2)\\
c_\mathrm{g}(\neg f) &= c_\mathrm{f}(f) & c_\mathrm{g}(f_1 \ \& \ f_2) &= c_\mathrm{f}(f_1) + c_\mathrm{f}(f_2)\\
c_\mathrm{g}(\forall_{V\in[b_1,b_2]} : f) &= 1 & c_\mathrm{g}(\forall^I_{V\leq p} : f) &= 1 + \textstyle\sum_{g\in I} c_\mathrm{i}(g)\\
c_\mathrm{g}(\forall_{V\in[p_1,p_2]} : f) &= 1 & c_\mathrm{i}((n, f, c)) &= c_\mathrm{f}(f)
\end{aligned}$$

Now we can define a relation which determines the maximum size of the runtime representation of a monitor encountered during its execution.

Definition 2. We define the relation $\multimap \subseteq \mathcal{M} \times \mathbb{N} \times \{\top, \bot\}^* \times \mathbb{N} \times \mathbb{N}$ inductively as follows:

$$\begin{aligned}
&M \multimap_{p,s,0} S' \leftrightarrow S' = c_\mathrm{m}(M)\\
&M \multimap_{p,s,(n+1)} S' \leftrightarrow\\
&\quad \exists R.\ (M \to_{p,s,s(p),R} M') \wedge (M' \multimap_{p+1,s,n} S) \wedge S' = \max\{c_\mathrm{m}(M), S\}
\end{aligned}$$

Since $\to$ is deterministic $M \multimap_{p,s,n} S$ uniquely determines S from M, p, s, and n. Essentially, $M \multimap_{p,s,n} S$ states that S is the maximum size of the representation of monitor m during the execution of n transitions over the stream s starting at position p. Our goal is to compute/bound the value of S by a static analysis, i.e., without having to actually perform the transitions. Towards this goal, we introduce in Sec. 3 an abstraction of the operational semantics for our abstraction of monitor specifications, which we call *evaluation vectors*. We then formalize the connection between our analysis and the above relation in Thm. 44. The rules of our abstracted operational semantics found in Sec. 3.2 (Def. 29) correspond directly to the rules of Figure 3. The relationship is not one to one, but all semantic properties of quantifiers are preserved. Non-quantifier rules are not important for memory analysis, so they have been dropped. Instance introduction is handled by a separate mechanism which exploits the translational symmetry of different stream positions.

2.4. Dominating Formula Transformation

A concept introduced in (Cerna et al., 2016c), *the Dominating Monitor/Formula*, allows us to restrict our analysis to quantified formulas whose variable intervals only depend on the outermost monitor variable, i.e. the size of every interval is the same for every value of the monitor variable.

Definition 3 (Dominating Monitor/Formula Transformation)**.** Let $\mathbb{A} = \mathbb{V} \rightarrow^{\text{part.}} \mathbb{N}$ be the domain of assignments that map variables to natural numbers. Then the *dominating monitor transformation* $D : \mathbb{M} \rightarrow \mathbb{M}$ respectively *formula transformation* $D' : \mathbb{F} \times \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{F}$ are defined as follows:

$$\begin{aligned}
D(\forall_{0\leq V} : F) &= \forall_{0\leq V} : D'(F, [V \mapsto 0], [V \mapsto 0]) \\
D'(@V, a_l, a_h) &= @V \\
D'(\neg F, a_l, a_h) &= \neg D'(F, a_l, a_h) \\
D'(F_1 \ \& \ F_2, a_l, a_h) &= D'(F_1, a_l, a_h) \ \& \ D'(F_2, a_l, a_h) \\
D'(F_1 \ \wedge \ F_2, a_l, a_h) &= D'(F_1, a_l, a_h) \ \wedge \ D'(F_2, a_l, a_h) \\
D'(\forall_{V\in[B_1,B_2]} : F, a_l, a_h) &= \forall_{V\in[h_L(B_1),h_H(B_2)]} : D'(F, a'_l, a'_h)
\end{aligned}$$

In the last equation we have $a'_l = a_l[V \mapsto h_L(B_1)]$, $a'_h = a_h[V \mapsto h_H(B_2)]$, $h_L(B_1) = \min\{[\![B_1]\!]^{a_l}, [\![B_1]\!]^{a_h}\}$, $h_H(B_2) = \max\{[\![B_2]\!]^{a_l}, [\![B_2]\!]^{a_h}\}$ where $[\![B]\!]^a$ denotes the result n of the evaluation of bound expression B for assignment a; actually, if B contains the monitor variable x, the result shall be the expression $x + n$ (we omit the formal details, see the example below).

Dominating monitors are used in the construction of *evaluation vectors* (see Def. 10).

Example 2. Consider the following monitor M:

$$\begin{aligned}
\forall_{0\leq x} : \forall_{y\in[x+1,x+5]} : ((\forall_{z\in[y,x+3]} : \neg@z \ \& \ @z) \ \& \ G(x, y)) \\
G(x, y) = \forall_{w\in[x+2,y+2]} : (\neg@y \ \& \ (\forall_{m\in[y,w]} : \neg@x \ \& \ @m))
\end{aligned}$$

The dominating form $D(M)$ of M is the following:

$$\begin{aligned}
\forall_{0\leq x} : \forall_{y\in[x+1,x+5]} : ((\forall_{z\in[x+1,x+3]} : \neg@z \ \& \ @z) \ \& \ G(x, y)) \\
G(x, y) = \forall_{w\in[x+2,x+7]} : (\neg@y \ \& \ (\forall_{m\in[x+1,x+7]} : \neg@x \ \& \ @m))
\end{aligned}$$

Notice that additional instances are needed for the evaluation of $D(M)$.

Fig. 4 illustrates the dominating monitor transformation: The left side shows the intervals of the inner quantifier for each value of y in the outer quantifier interval; notice that the values increase with respect to the value of y. The right-hand side represents the interval structure of the dominating monitor; the values of the inner quantifier's interval are constant, invariant of the value of y. Essentially the future most position is fixed in the dominating monitor, where as, in the original monitor, the future most position is dependent on y.

The relationship, in terms of the maximum size of instance sets, between a monitor M and its dominating form $D(M)$ is summarized in the following theorem.

Theorem 4. *Let $M \in \mathbb{M}$. Then for all $p, n, S, S' \in \mathbb{N}$ and $s \in \{\top, \bot\}^\omega$ such that $T(M) \multimap_{p,s,n} S$ and $T(D(M)) \multimap_{p,s,n} S'$, we have $S \leq S'$.*

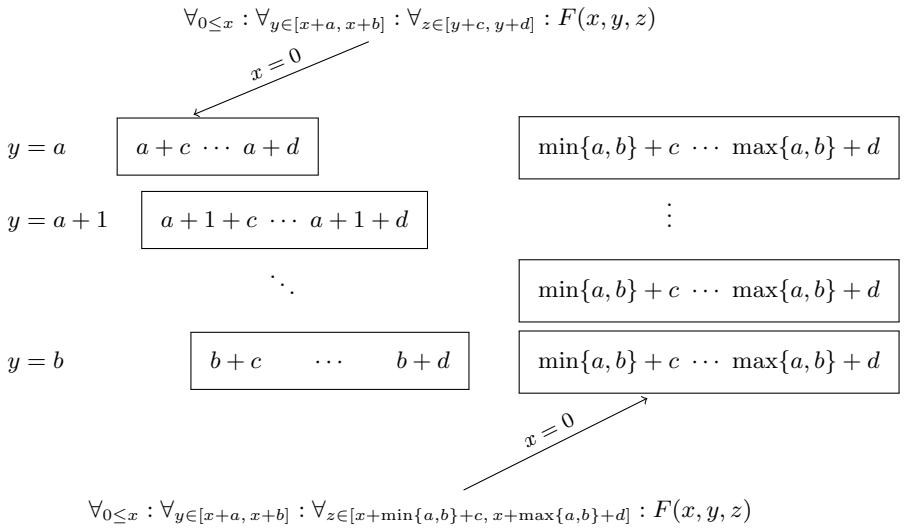

Fig. 4. Illustration of the dominating monitor transformation.

The correctness of this theorem follows from Def. 2 and 3. Clearly, if $M = D(M)$, i.e., the monitor is already in its dominating form, then we have $S = S'$.

3. Quantifier Vectors and their Evaluation

In (Cerna et al., 2016c) the logical structure of the monitor specifications was considerably abstracted. However, when dealing with quantifiers which are nested more than two levels deep, that abstraction becomes quite cumbersome. This is why we considered an alternative abstraction in (Cerna et al., 2016a). We referred to this abstraction as *quantifier trees*. Quantifier trees drop the quantifier free parts of the monitor specification entirely and keep the structures which influence space complexity to a significant extent (future looking quantifiers). A less express subset of quantifier tree, *quantifier vectors*, can be used to derive bounds for the most restricted monitors. To aid our abstract evaluation mechanism, we define a type of labelled quantifier vector called an *evaluation vectors*. When working with evaluation vectors we assume that they are an abstraction of a dominating monitor and thus we can even drop the variable names.

3.1. Quantifier and Evaluation Vectors

In this section we build all abstractions necessary for deriving the results of later sections. Though, for the algorithm, we will need to add additional labels to distinguish between the infinite and finite memory case.

Definition 5 (Quantifier Trees)**.** A *quantifier tree* is inductively defined to be either $\emptyset$ or a tuple of the form (y, b_1, b_2, Q) where $y \in V$, $b_1, b_2 \in B$ and Q is a set of quantifier trees. Let $\mathbb{QT}$ be the set of all quantifier trees.

Definition 6 (Quantifier Tree Transformation)**.** We define the *quantifier tree transformation* $QT : \mathbb{M} \to \mathbb{QT}$, respectively $QT : \mathbb{F} \to \mathbb{QT}$, recursively as follows:

$$\begin{aligned} QT(\forall_{0\leq V} : F) &= (V, 0, 0, QT(F)) & QT(F\&G) &= QT(F) \cup QT(G) \\ QT(F \wedge G) &= QT(F) \cup QT(G) & QT(\neg F) &= QT(F) \\ QT(\forall_{V\in[B_1,B_2]} : F) &= (V, B_1, B_2, QT(F)) & QT(@V) &= \emptyset \end{aligned}$$

We do not distinguish between sequential and parallel conjunction in quantifier trees being that their effect on memory is dependent on the stream they are evaluated over, thus, we just take the worst case scenario of parallel conjunction.

Definition 7 (Quantifier Tree Node)**.** Let $q = (y, b_1, b_2, Q) \in \mathbb{QT}$. Then n is a *quantifier tree node of* q if for some $b \in \{0, 1\}$, $n = [(y, b_1, b_2, b)]$.

Definition 8 (Quantifier Tree Path)**.** Let $q = (y_1, b_1^1, b_2^1, Q) \in \mathbb{QT}$. Then we define inductively p as a *quantifier tree path of* q, if $p = []$ or

$$p = \left[(y_1, b_1^1, b_2^1, b_1), (y_2, b_1^2, b_2^2, b_2), \cdots, (y_n, b_1^n, b_2^n, b_n)\right]$$

and for some $q' \in Q$, $p' = \left[(y_2, b_1^2, b_2^2, b_2), \cdots, (y_n, b_1^n, b_2^n, b_n)\right]$ is a quantifier tree path of q'. Let $\mathbb{QP}(q)$ denote the set of quantifier tree paths of a quantifier tree q.

Though not said explicitly in the definition, if a quantifier tree path p has length 1 then it is also a quantifier tree node. The fourth position of a quantifier tree node is the labelling needed in Sec. 3.2. We will work with a degenerate case of quantifier trees, namely, *quantifier vectors*.

Definition 9 (Quantifier Vectors)**.** A *quantifier vector* is inductively defined to be either $\emptyset$ or a tuple of the form (y, b_1, b_2, Q) where $y \in V$, $b_1, b_2 \in B$ and Q is a set of quantifier vectors s.t. $|Q| \leq 1$. Let $\mathbb{QV}$ be the set of all quantifier vectors. We define *the length* $|v|$ *of a quantifier vector* v, as $|v| = 0$ if $v = \emptyset$, $|v| = 1$ if $v = (y, b_1, b_2, \emptyset)$, or $|v| = 1 + |q|$ if $v = (y, b_1, b_2, Q)$ and $q \in Q$.

Though the recursive definition is cleaner, we will use the quantifier tree path representation of a quantifier vector being that the structure we need to work with is explicitly represented. The following concept considers only monitors whose bounds are expressed in terms of the stream variable, i.e. dominating monitors (Definition 3). The subsequently derived space complexity results are optimal for specific dominating monitors and represent upper bounds for pre-transformation monitors.

Definition 10 (Evaluation Vector)**.** Let $M \in \mathbb{M}$ such that $QT(M) \in \mathbb{QV}$. Then an *evaluation vector* of M is a quantifier tree path $e \in \mathbb{QP}(QT(D(M)))$. Since quantifier variables no longer play a role in interval evaluation, we use the short hand $\left[(c_1^1, c_2^1, b_1), \cdots, (c_1^n, c_2^n, b_n)\right]$ for $\left[(y_2, x + c_1^1, x + b_2^1, b_1), \cdots, (y_n, x + c_1^n, x + c_2^n, b_n)\right]$. Let $\mathbb{EV}$ be the set of all evaluation vectors.

Definition 11. Let $v = \left[(c_1^1, c_2^1, b_1), \cdots, (c_1^n, c_2^n, b_n)\right] \in \mathbb{EV}$. We define $v(i) = (c_1^i, c_2^i, b_i)$, $v(i, 1) = c_1^i$, $v(i, 2) = c_2^i$, and $v(i, 3) = b_i$. When $v(i, 3) = 0$ for all $1 \leq i \leq |v|$ we call v *proper*.

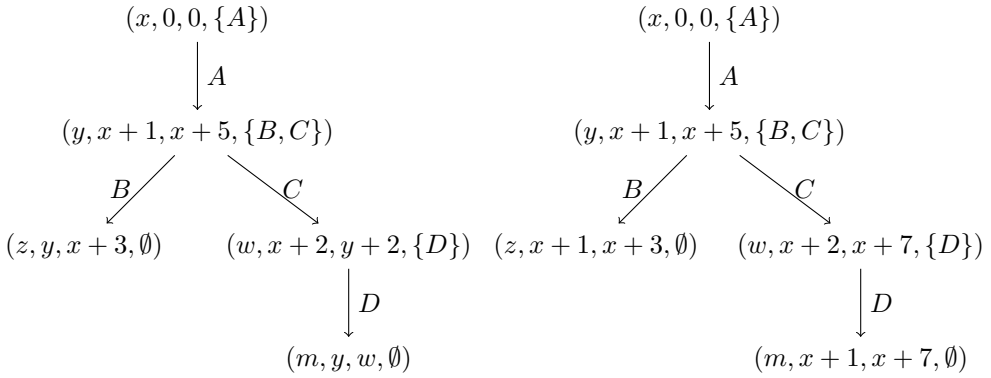

Fig. 5. Quantifier trees for the monitor of Ex. 2.

Definition 12. Given $v = \left[(c_1^1, c_2^1, b_1), \cdots, (c_1^n, c_2^n, b_n)\right] \in \mathbb{EV}$, the *state string of* v, $ss(v)$, is the binary word $b_1 b_2 \cdots b_n$. By $ss(v, i)$ for $i \in \{0, 1\}$ we denote the number of positions in state strings of v which have value i.

Example 3. Fig. 5 depicts the quantifier tree $QT(M)$ on the left side and $QT(D(M))$ on the right side; $QT(D(M))$ consists of two quantifier vectors:

$$(y, x+1, x+5, \{(z, x+1, x+3, \emptyset)\})$$

and

$$(y, x+1, x+5, \{(w, x+2, x+7, \{(m, x+1, x+7, \emptyset)\})\})$$

Possible evaluation vectors for the former are $[(1, 5, 0), (1, 3, 0)]$ or $[(1, 5, 1), (1, 3, 0)]$.

Definition 13. Let $v, w \in \mathbb{EV}$. The *concatenation of* v *and* w, $v \otimes w$, is defined as the vector $c \in \mathbb{EV}$ such that, for $1 \leq i \leq |v|$, $v(i) = c(i)$ and for $1 \leq j \leq |w|$, $w(|v| + j) = c(|v| + j)$.

Definition 14. Let $v, w_0, w, w_1 \in \mathbb{EV}$ s.t. $v = w_0 \otimes w \otimes w_1$. We refer to w as a *sub-evaluation vector of* v. If $0 < |w_0| + |w_1|$ then w is called *proper*. The set of sub-evaluation vectors of v is $\mathbb{EV}(v)$.

Example 4. Let us consider the evaluation vector $v = [(1, 5, 0), (2, 7, 0), (1, 7, 0)]$. We can write v using concatenation as $v = [(1, 5, 0)] \otimes [(2, 7, 0)] \otimes [(1, 7, 0)]$. A proper sub-evaluation vector of v is $[(2, 7, 0)]$ where $w_0 = [(1, 5, 0)]$ and $w_1 = [(1, 7, 0)]$.

Definition 15. Let $v \in \mathbb{EV}$. We refer to v as *uniform standard* if there exist $a, b \in \mathbb{Z}$, where $a \leq b$, such that for all $w \in \mathbb{EV}(v)$, $w(1, 1) = a$ and $w(1, 2) = b$. Let $\mathbb{US}$ be the set of uniform standard evaluation vectors and $\mathbb{US}(v)$ be the uniform standard evaluation vectors of a vector v.

Concatenation is important because it allows us to decompose complex evaluation vectors into *sections* which are in $\mathbb{US}$.

Definition 16. Let $v \in \mathbb{EV}$ and $s \in \mathbb{US}(v)$. s is a *section* of v if for every non-empty $w \in \mathbb{EV}(v)$, we have $s \otimes w \notin \mathbb{US}(v)$ and $w \otimes s \notin \mathbb{US}(v)$. Let $\mathbb{S}(v)$ be the multi-set of sections of a vector v.

Definition 17. Let $v \in \mathbb{EV}$. We define $<_v$ as a total ordering on $\mathbb{S}(v)$ such that for $s, s' \in \mathbb{S}(v)$, $s <_v s'$ iff there exist $v_0, v_1, v_2 \in \mathbb{EV}(v)$ (not necessarily non-empty) such that $v = v_0 \otimes s \otimes v_1 \otimes s' \otimes v_2$. For any vector $v \in \mathbb{EV}$ $[]$ is $(v, 0)$-*perfect*. The minimum element of $<_v$ is $(v, 1)$-*perfect*. For $w \in \mathbb{EV}(v)$ and $m \in \mathbb{N}$, w is $(v, m+1)$-*perfect* if $w = h \otimes s$, for some $h \in \mathbb{EV}(v)$, such that h is (v, m)-*perfect*, $s \in \mathbb{S}(v)$, and $v = w \otimes v_2$, for some $v_2 \in \mathbb{EV}(v)$.

Definition 18. Let $v \in \mathbb{EV}$. We refer to v as *top-free* if there exists $a \in \mathbb{Z}$ such that for all $w \in \mathbb{EV}(v)$ we have $w(1,1) = a$ and $a \leq w(1,2)$. Let $\mathbb{TF}$ be the set of top-free vectors and $\mathbb{TF}(v)$ the set of top-free vectors of a vector v.

Example 5. Let us consider the vector

$$v = [(1,5,0), (1,7,0)(1,7,0), (1,8,0), (1,3,0), (1,3,0), (1,8,0)]$$

This vector is a member of $\mathbb{TF}$ and has the following sections contained in the multi-set $\mathbb{S}(v)$:

$$\mathbb{S}(v) = \left\{ \begin{array}{ll} [(1,5,0)] & [(1,7,0)(1,7,0)] \\ [(1,8,0)] & [(1,3,0),(1,3,0)] \\ [(1,8,0)] & \end{array} \right\}$$

The ordering of these sections is as follows:

$$[(1,5,0)] <_v [(1,7,0)(1,7,0)] <_v [(1,8,0)] <_v [(1,3,0),(1,3,0)] <_v [(1,8,0)]$$

From the order we can derive the perfect vectors of v. The $(v,1)$-*perfect* vector is $[(1,5,0)]$ and the $(v,2)$-*perfect* vector $[(1,5,0),(2,7,0)(2,7,0)]$. The $(v,5)$-*perfect* vector is v itself. Every member of $\mathbb{S}(v)$ is also a member of $\mathbb{US}$.

Perfect vectors are interesting being that they allow us to talk about sub-vectors in a way which preserves the ordering of the original vector. They also allow us to define two essential subsets of evaluation vectors based on the top-free vector construction.

Definition 19. Let $v \in \mathbb{EV}$ be top-free and $1 \leq i < |\mathbb{S}(v)|$ and $w, h \in \mathbb{EV}(v)$ be (v,i)-*perfect* and $(v, i+1)$-*perfect*, respectively. We refer to v as *standard* if $w(|w|, 2) \leq h(|h|, 2)$. We refer to v as *inverse standard* if $w(|w|, 2) \geq h(|h|, 2)$. Let $\mathbb{SE}$ be the set of standard vectors and $\mathbb{IS}$ be the set of inverse standard vectors. Let $\mathbb{SE}(v)$ be the set of standard vectors of a vector v and $\mathbb{IS}(v)$ be the set of inverse standard vectors of v.

Example 6. Let us consider the vector

$$v = [(1,5,0), (1,7,0)(1,7,0), (1,8,0), (1,3,0), (1,3,0), (1,8,0)]$$

which is a member of $\mathbb{TF}$. The following two vectors are constructable from $\mathbb{S}(v)$:

$$v_s = [(1,3,0), (1,3,0), (1,5,0), (1,7,0)(1,7,0), (1,8,0), (1,8,0)] =$$
$$[(1,3,0), (1,3,0)] \otimes [(1,5,0)] \otimes [(1,7,0)(1,7,0)] \otimes [(1,8,0)] \otimes [(1,8,0)]$$

$$v_i = [(1,8,0), (1,8,0), (1,7,0)(1,7,0), (1,5,0), (1,3,0), (1,3,0)] =$$
$$[(1,8,0)] \otimes [(1,8,0)] \otimes [(1,7,0)(1,7,0)] \otimes [(1,5,0)] \otimes [(1,3,0), (1,3,0)]$$

Obviously be Def. 19, $v_s \in \mathbb{SE}$ and $v_i \in \mathbb{IS}$. This property is formalized in the following lemma.

Lemma 20. *Let $v \in \mathbb{TF}$. Then there exist vectors $v', v'' \in \mathbb{TF}$ such that $\mathbb{S}(v) = \mathbb{S}(v') = \mathbb{S}(v'')$ and $v' \in \mathbb{SE}$ and $v'' \in \mathbb{IS}$.*

Proof. We choose v' to be the vector with the ordering relation $s, s' \in \mathbb{S}(v')$, $s <_{v'} s'$ iff $s(1,2) < s'(1,2)$ and v'' with the ordering relation $s, s' \in \mathbb{S}(v')$, $s <_{v''} s'$ iff $s(1,2) > s'(1,2)$. □

This covers the basic theory of our abstraction, but not how these representations of monitors are evaluated. In the next subsection we use the additional labelling of quantifier tree nodes to evaluate the evaluation vectors. We derive a set of rules simulating rules $Q1-7$ from Fig. 3 which use the label to mark positions which have been evaluated. The label is specifically used by a *splitting operation* (splitting a vector into its evaluated and to be evaluated parts). Splitting is a method of generating the new vectors resulting from evaluation. We need one additional step in order to choose the correct vectors resulting from splitting or else our evaluation will not match the evaluation of the core language.

3.2. Evaluation

In this section we first introduce the concept of splitting evaluation vectors and then use the concept to evaluate evaluation vectors in a similar way as the operational semantics evaluates monitor specifications.

Definition 21. Let $v = h \otimes [(a,b,0)] \otimes w \in \mathbb{EV}$, where $ss(h,1) = |h|$ and $ss(w,0) = |w|$. A *split* $spt(v)$ *of* v denotes the following set of evaluation vectors:

- If $ss(v,1) < |v|$ then
  - if $w \neq []$ then
    - if $a < b$, then $spt(v) = \{h \otimes [(a,b,1)] \otimes w, h \otimes [(a+1,b,0)] \otimes w\}$
    - if $a = b$, then $spt(v) = \{h \otimes [(a,b,1)] \otimes w\}$
    - if $a \geq b$, then $spt(v) = \{[]\}$.
  - if $w = []$ then
    - if $a < b$ then $spt(v) = \{h \otimes [(a+1,b,0)]\}$
    - if $a = b$ then $spt(v) = \{h \otimes [(a,b,1)]\}$
    - if $a \geq b$, then $spt(v) = \{[]\}$.
- If $ss(v,1) = |v|$ then $spt(v) = \{v\}$.

Definition 22. Let $v \in \mathbb{EV}$ be proper and $n \in \mathbb{N}$. We define the *n-iterated split* $SI(n,v)$ of v inductively as follows:

$$SI(n+1,v) = \left(\bigcup_{w \in SI(n,v)} spt(w) \right) \cup SI(n,v)$$
$$SI(0,v) = \{v\}$$

Example 7. Fig. 6 shows the iterated split operation applied up to the third level to the vector $[(0,3,0),(0,3,0)]$. The iterated split $SI(3,v)$ is the union of all levels of the given tree. The gray nodes of the tree are evaluation vectors of which the split operator cannot be applied to, i.e. vectors v such that $ss(v,1) = |v|$.

Lemma 23. *There exists $n \in \mathbb{N}$ such that $SI(n+1,v) = SI(n,v)$.*

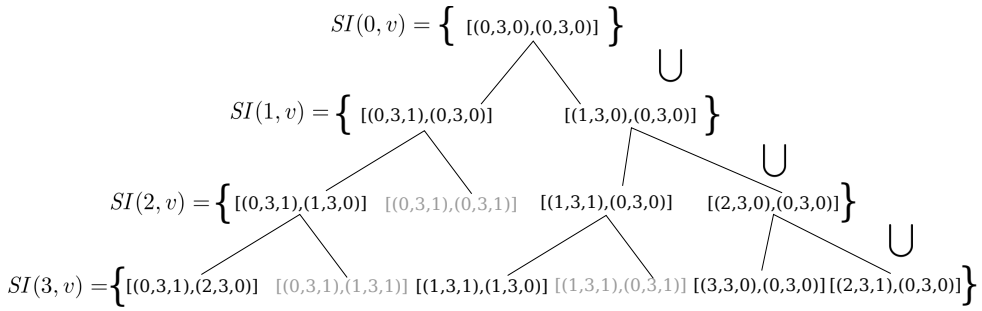

Fig. 6. Example of iterated splitting applied to a uniform evaluation vector.

Proof (sketch). Trivial from the definition of split. □

Definition 24. Let $v \in \mathbb{EV}$ be proper. We define the *split closure* $SC_v = SI(n, v)$ of v, where n is the smallest $n \in \mathbb{N}$ such that $SI(n+1, v) = SI(n, v)$.

Lemma 25. *Let $v \in \mathbb{EV}$ be proper. Then for every vector $w \in SC_v$, there exist $k, m \in \mathbb{N}$ such that $m + k = |v|$ and $ss(w) = \overbrace{1 \cdots 1}^{m}\overbrace{0 \cdots 0}^{k}$.*

Definition 26. Let $v \in \mathbb{EV}$ be proper. The *degenerate vector set* DV_v of v is defined as follows:

$$DV_v = \{w \in SC_v | spt(w) = \{[]\}\}$$

Definition 27. Let $v \in \mathbb{EV}$ be proper. The set D_v of *derived vectors* of v is $D_v = (SC_v \setminus DV_v)$.

Example 8. Let us consider $v = [(0,1,0),(0,2,0)]$. Then

$$SI(5, v) =$$

$$\left\{ \begin{array}{cccc} [(0,1,0),(0,2,0)] & [(1,1,0),(0,2,0)] & [(0,1,1),(0,2,0)] & [(1,1,1),(0,2,0)] \\ [(0,1,1),(0,2,1)] & [(0,1,1),(1,2,0)] & [(1,1,1),(0,2,1)] & [(1,1,1),(1,2,0)] \\ [(0,1,1),(1,2,1)] & [(0,1,1),(2,2,0)] & [(1,1,1),(2,2,0)] & [(1,1,1),(1,2,1)] \\ [(0,1,1),(2,2,1)] & [(1,1,1),(2,2,1)] & [\ ] & \end{array} \right\}$$

$$DV_v = \left\{ \begin{array}{cccc} [(0,1,1),(0,2,1)] & [(1,1,1),(0,2,1)] & [(0,1,1),(1,2,1)] & [(1,1,1),(1,2,1)] \\ [(0,1,1),(2,2,1)] & [(1,1,1),(2,2,1)] & [\ ] & \end{array} \right\}$$

$$D_v = \left\{ \begin{array}{cccc} [(0,1,0),(0,2,0)] & [(1,1,0),(0,2,0)] & [(0,1,1),(0,2,0)] & [(1,1,1),(0,2,0)] \\ [(0,1,1),(1,2,0)] & [(1,1,1),(1,2,0)] & [(0,1,1),(2,2,0)] & [(1,1,1),(2,2,0)] \end{array} \right\}$$

The derived set D_v represents instances which need to be kept in memory during the evaluation of a monitor specification. However, the size of the derived set does not directly correspond to the number of instance in memory at a given time, because not

all of the instances in the set need to be in memory at the same time. For instance, $[(1,1,0),(0,2,0)]$ and $[(1,1,1),(1,2,0)]$ cannot be in memory at the same time because they state contradictory states of evaluation. The derived set is constructed specifically for defining the domain of our evaluation function.

Note that splitting divides the vector into two parts, a part with one in the third component and a part with zeroes in the third component, see Lem. 25. The following lemma addresses this and simplifies the definition of evaluation.

Lemma 28. *Let $v \in \mathbb{EV}$ be non-empty and proper. For every vector $w \in D_v$, there exist $\mathbf{1}_w, \mathbf{h}_w, \mathbf{0}_w \in \mathbb{EV}(v)$, such that $w = \mathbf{1}_w \otimes \mathbf{h}_w \otimes \mathbf{0}_w$, where $\mathbf{1}_w(i,3) = 1$, $1 \leq i \leq |\mathbf{1}_w|$, $\mathbf{h}_w(1,3) = 0$ and $|\mathbf{h}_w| = 1$, and $\mathbf{0}_w(i,3) = 0$, $1 \leq i \leq |\mathbf{0}_w|$. Both $\mathbf{0}_w$ and $\mathbf{1}_w$ can be the empty vector, but $\mathbf{h}_w$ is always non-empty.*

Proof. Follows, almost directly, from Lem. 25. □

Definition 29. Let $v \in \mathbb{EV}$ be non-empty and proper, $s \in \mathbb{N}$, and $w \in D_v$. We define the sets $e(s,w,v)$ and $\overline{e}(s,w,v)$ as follows:

E1) If $s < \mathbf{h}_w(1,1)$, then

$$e(s,w,v) = \{w\}$$
$$\overline{e}(s,w,v) = \emptyset$$

E2) If $\mathbf{h}_w(1,1) \leq s < \mathbf{h}_w(1,2)$ and $\mathbf{0}_w \neq []$, then

$$e(s,w,v) = \left(\bigcup_{i=\mathbf{h}_w(1,1)}^{s} e(s, \mathbf{1}_w \otimes v_i \otimes \mathbf{0}_w, v)\right) \cup \{\mathbf{1}_w \otimes w_{s+1} \otimes \mathbf{0}_w\}$$
$$\overline{e}(s,w,v) = \left(\bigcup_{i=\mathbf{h}_w(1,1)}^{s} \overline{e}(s, \mathbf{1}_w \otimes v_i \otimes \mathbf{0}_w, v)\right)$$

where $v_i = [(i, \mathbf{h}_w(1,2), 1)]$ and $w_{s+1} = [(s+1, \mathbf{h}_w(1,2), 0)]$.

E3) If $\mathbf{h}_w(1,2) \leq s$ and $\mathbf{0}_w \neq []$,then

$$e(s,w,v) = \left(\bigcup_{i=\mathbf{h}_w(1,1)}^{\mathbf{h}_w(1,2)} e(s, \mathbf{1}_w \otimes v_i \otimes \mathbf{0}_w, v)\right)$$
$$\overline{e}(s,w,v) = \left(\bigcup_{i=\mathbf{h}_w(1,1)}^{\mathbf{h}_w(1,2)} \overline{e}(s, \mathbf{1}_w \otimes v_i \otimes \mathbf{0}_w, v)\right)$$

where $v_i = [(i, \mathbf{h}_w(1,2), 1)]$.

E4) If $\mathbf{h}_w(1,1) \leq s < \mathbf{h}_w(1,2)$ and $\mathbf{0}_w = []$, then

$$e(s,w,v) = \{\mathbf{1}_w \otimes w_{s+1} \otimes \mathbf{0}_w\}$$
$$\overline{e}(s,w,v) = \bigcup_{i=\mathbf{h}_w(1,1)}^{s} \{\mathbf{1}_w \otimes v_i \otimes \mathbf{0}_w\}$$

where $w_{s+1} = [(s+1, \mathbf{h}_w(1,2), 0)]$, and $v_i = [(i, \mathbf{h}_w(1,2), 1)]$.

E5) If $\mathbf{h}_w(1,2) \leq s$ and $\mathbf{0}_w = []$, then

$$e(s,w,v) = \emptyset$$
$$\overline{e}(s,w,v) = \bigcup_{i=\mathbf{h}_w(1,1)}^{\mathbf{h}_w(1,2)} \{\mathbf{1}_w \otimes v_i \otimes \mathbf{0}_w\}$$

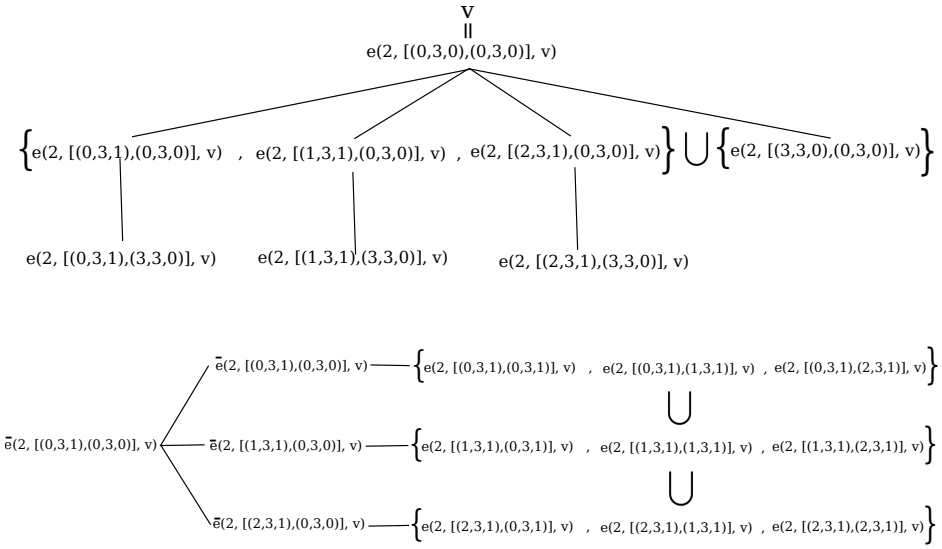

Fig. 7. Application of Def. 29 to the evaluation vector $[(0,3,0),(0,3,0)]$

Example 9. In Fig. 7 we provide the derivation tree of both the function e and the function $\overline{e}$ for the value 2 applied to the vector $[(0,3,0),(0,3,0)]$. For

$$e(2,[(0,3,0),(0,3,0)],[(0,3,0),(0,3,0)])$$

we did not completely expand the set being that it is too big. However, we show the result of the first two steps. For $\overline{e}(2,[(0,3,0),(0,3,0)],[(0,3,0),(0,3,0)])$, the union of the leaves represents the content of the set.

We discussed at length in section Sec. 2.3 the relationship between Def. 29 and Fig. 3. We have now formally introduced both concepts and can draw deeper connections between the two. We will give a direct comparision between the **E** rules and the Q rules. Let us start with the rule **E1** and $Q3$. The rule states that evaluation of an evaluation vector w at a position s such that $s < \mathbf{h}_w(1,1)$ is not possible. The rule $Q3$ similarly states that a formula transition does nothing when the current position is lower than the lower bound of the quantifier. The relationship between **E2** and $Q4$ is more interesting. The rule **E2** states that evaluation of an evaluation vector w at a position s, such that $\mathbf{h}_w(1,1) < s < \mathbf{h}_w(1,2)$, constructs $s - \mathbf{h}_w(1,1) + 1$ new instances and evaluates them individually. Also the lower bound of $\mathbf{h}_w$ is replaced by $s+1$ to represent the partial evaluation. The rule $Q4$ similarly states that an instance set ought to be constructed for the formula being evaluated by instantiating the quantifier's variable with the values between p_1 and p. replace the lower bound of the quantifier with the next position. Rule **E3** and **E4** are equivalent to the continuation rule of the operational semantics, **Q7**. However, our abstraction ignores truth and it is likely that the result of an application of the rules **E3** and **E4** would produce a state which does not correspond to the results of the operational semantics, i.e. **Q5** or **Q6** would have been applied instead of **Q7**. This property highlights how this abstraction over approximates the actual number of instances in memory at runtime. The final rule **E5** implies that evaluation is complete and enough information has been gathered to apply either **Q5** or **Q6**.

Lemma 30. *Let $v \in \mathbb{EV}$ be non-empty and proper, $w \in D_v$ and $0 \leq m$. Then $E(m, w, v) \equiv E(m, \mathbf{h}_w \otimes \mathbf{0}_w, v)$.*

Proof. Follows from Def. 29. □

Lemma 31. *Let $v \in \mathbb{EV}$ be non-empty and proper, $w \in D_v$ and $0 \leq m$, then $e(m, w, v) \subseteq D_v$ and $\overline{e}(m, w, v) \subseteq DV_v$.*

Proof. $e(m, w, v)$ represents multiple applications of the split operator. The split operator results in vectors which are either in D_v or DV_v. By Def. 29, we know that for every $v' \in e(m, w, v)$, $ss(v', 1) \leq |v'|$ and thus $v' \notin DV_v$. Also, for every $v' \in \overline{e}(m, w, v)$, $ss(v', 1) = |v'|$ and thus $v' \in DV_v$. And thus, from these facts the lemma follows. □

We can also consider the application of Def. 29 to sets of vectors.

Definition 32. Let $v \in \mathbb{EV}$ be non-empty and proper, V a set of vectors, such that $V \subseteq D_v$, and $0 \leq m$. Then we define $e(m, V, v) = \bigcup_{w \in V} e(m, w, v)$.

Definition 33. Let $v \in \mathbb{EV}$ be non-empty and proper, $w \in D_v$, and $0 \leq m$. Then we define

$$E(m, w, v) = \begin{cases} e(m, E(m-1, w, v), v) \text{ if } v(1,1) < m \\ \quad e(v(1,1), m, v) \qquad \text{otherwise} \end{cases}$$

$$\overline{E}(m, w, v) = \bigcup_{i=v(1,1)}^{m} \overline{e}(i, w, v)$$

Lemma 34. *Let $v \in \mathbb{EV}$ be non-empty and proper, $w \in D_v$, and $0 \leq m$. Then $E(m, w, v) \subseteq D_v$ and $\overline{E}(m, w, v) \subseteq DV_v$.*

Proof. Similar to the proof of Lem. 31. □

Within a set of vectors $V \subseteq D_v$ for some non-empty and proper $v \in \mathbb{EV}$, it is of importance (especially for the proofs of theorems in the next section) to know how many vectors $w \in V$ exists such that $w(i) = \mathbf{h}_w$.

Definition 35. Let $v \in \mathbb{EV}$ be non-empty and proper, and $V \subseteq D_v$ be a set of vectors. We define $\{V\}^i$ for all $1 \leq i \leq |v|$ as $\{V\}^i = \{w \in V \mid \mathbf{h}_w = w(i)\}$. The size $|V|^i$ of $\{V\}^i$ is defined as $|V|^i = |\{V\}^i|$.

Example 10. Let us consider the evaluation of the vector $v = [(0,5,0),(0,6,0),(0,7,0)]$.

$$E(0, v, v) = \left\{ \begin{array}{l} [(1,5,0),(0,6,0),(0,7,0)] \ [(0,5,1),(1,6,0),(0,7,0)] \\ [(0,5,1),(0,6,1),(1,7,0)] \end{array} \right\}$$

$$E(1, v, v) = \left\{ \begin{array}{l} [(2,5,0),(0,6,0),(0,7,0)] \ [(1,5,1),(2,6,0),(0,7,0)] \\ [(1,5,1),(0,6,1),(2,7,0)] \ [(1,5,1),(1,6,1),(2,7,0)] \\ [(0,5,1),(2,6,0),(0,7,0)] \ [(0,5,1),(1,6,1),(2,7,0)] \\ [(0,5,1),(0,6,1),(2,7,0)] \end{array} \right\}$$

$$E(2,v,v) = \left\{ \begin{array}{ll} [(3,5,0),(0,6,0),(0,7,0)] & [(2,5,1),(3,6,0),(0,7,0)] \\ [(2,5,1),(0,6,1),(3,7,0)] & [(2,5,1),(1,6,1),(3,7,0)] \\ [(2,5,1),(2,6,1),(3,7,0)] & [(1,5,1),(3,6,0),(0,7,0)] \\ [(1,5,1),(2,6,1),(3,7,0)] & [(1,5,1),(0,6,1),(3,7,0)] \\ [(1,5,1),(1,6,1),(3,7,0)] & [(0,5,1),(3,6,0),(0,7,0)] \\ [(0,5,1),(2,6,1),(3,7,0)] & [(0,5,1),(1,6,1),(3,7,0)] \\ [(0,5,1),(0,6,1),(3,7,0)] & \end{array} \right\}$$

Take note of the growth of $|E(i,v,v)|^{j+1}$, for $j \in \{0,1,2\}$ and $0 \leq i$:

		i						
		0	1	2	3	4	5	6
	0	1	1	1	1	1	0	0
j	1	1	2	3	4	5	6	0
	2	1	4	9	16	25	36	42

Essentially, $|E(i,v,v)|^1 = O(i)$ and $|E(i,v,v)|^2 = O(i^2)$. This observation is an essential part of the results in the next section.

Lemma 36. *Let $v \in \mathbb{EV}$ be non-empty and proper, $l \in \mathbb{N}$, where $1 \leq l < |\mathbb{S}(v)|$, and $w,h \in \mathbb{EV}(v)$ where w is $(v,l+1)$-perfect and h is (v,l)-perfect. Then for $m,q \in \mathbb{N}$, such that $m \leq w(|w|,2)$ and $|h| \leq q < |w|$, $|E(m,v,v)|^q = |E(m,w,w)|^q$ holds.*

Proof. Follows from Def. 29, Def. 35 and Lem. 25. Any vector $r \in E(m,v,v)$ such that $ss(r,1) \geq |w|$ will not be counted by $|E(m,v,v)|^q$. Thus we can ignore them. This allows us to only consider the vector w itself. □

Lemma 37. *Let $v \in \mathbb{EV}$ be non-empty and proper, $l \in \mathbb{N}$, where $1 \leq l < |\mathbb{S}(v)|$, $s \in \mathbb{S}(v)$, and $w,h \in \mathbb{EV}(v)$ where w is $(v,l+1)$-perfect, h is (v,l)-perfect, and $w = h \otimes s$. Then for $m,q \in \mathbb{N}$, such that $m \leq w(|w|,2)$ and $|h| \leq q < |w|$. We have*

$$|E(m,v,v)|^q = |\overline{E}(m,h,h)| \cdot |E(m,s,s)|^{|s|}$$

Proof. For every $g \in \overline{E}(m,h,h)$, $ss(w,1) = |h|$. If we were to extend each of these evaluation vectors with every evaluation vector in $E(m,s,s)$, then we would get $E(m,w,w)$. By Lem. 36 we get $|E(m,v,v)|^q = |E(m,w,w)|^q$. Putting everything together we get the lemma. □

Corollary 38. *Let $v \in \mathbb{EV}$ be non-empty and proper and $l \in \mathbb{N}$, where $0 \leq l < |\mathbb{S}(v)|$. Let $w,h \in \mathbb{EV}(v)$, where w is (v,l)-perfect, and $v = w \otimes h$. Let $q,m \in \mathbb{N}$, where $|w| < q \leq |v|$ and $m \leq h(1,2)$. Then*

$$|E(m,v,v)|^q = |\overline{E}(m,w,w)| \cdot |E(m,h,h)|^{q'}$$

where $q' = q - |w|$.

Proof. Apply Lem. 36 in the inverse direction. □

Cor. 38 can easily be applied within a single section of an evaluation, which we will use in the following sections.

Lemma 39. *Let $v \in \mathbb{EV}$ be non-empty and proper, V be a set of vectors, s.t. $V \subseteq D_v$,and $m \geq 0$. Then the following holds:*

$$|E(m,v,v)| = \sum_{i=1}^{|v|} |E(m,v,v)|^i \tag{1a}$$

$$E(m,V,v) = \bigcup_{i=1}^{|v|} E(m,\{V\}_1^i,v) = \bigcup_{i=1}^{|v|} \bigcup_{w\in\{V\}^i} E(m,w,v) \tag{1b}$$

$$|E(m,V,v)| = \sum_{i=1}^{|v|} |E(m,\{V\}^i,v)| = \sum_{i=1}^{|v|} \sum_{w\in\{V\}^i} |E(m,w,v)| \tag{1c}$$

$$|E(m,V,v)|^i = \sum_{j=1}^{|v|} |E(m,\{V\}^j,v)|^i = \sum_{j=1}^{|v|} \sum_{w\in\{V\}^j} |E(m,w,v)|^i \tag{1d}$$

$$|E(m,V,v)|^i = |\bigcup_{j=1}^{i} E(m,\{V\}^j,v)|^i \tag{1e}$$

Lem. 39 was used heavily when proving Thm. 47 in (Cerna et al., 2016a). The proof heavily relied on rewriting of the evaluation function in order to reach a statement justified by the induction hypothesis. The statements in Lem. 39 can be proven from the above definitions concerning the evaluation function. Def. 33 is very close to our concept of runtime representation size, but only for a single step. We need to add new instances and evaluate those instances as well.

Definition 40. Let $s = [(a,b,0)] \in \mathbb{EV}$ and $n \in \mathbb{N}$. The *n-step-pair $s : n$ of s* is defined as $s : n = [(a+n, b+n, 0)]$.

Definition 41. Let $v, v' \in \mathbb{EV}$ be non-empty and proper, $s \in \mathbb{S}(v)$, s.t. $v = v' \otimes s$, and $n \in \mathbb{N}$. We define the *step-pair generation function* $st(n,v)$ inductively as follows:

$$st(n, v' \otimes s) = st(n,v') \otimes s : n$$
$$st(n,[]) = [] .$$

We write $st(n,v)$ as $v : n$.

Note that in Def. 41, the induction is not over n, but rather over the total order $<_v$.

Lemma 42. *Let $v \in \mathbb{EV}$ be non-empty and proper and $m, n \in \mathbb{N}$. Then $|E(m,v,v)| = |E(n+m, v:n, v:n)|$.*

Proof. Essentially, adding one to every position in a vector v is the same as evaluating the vector at a position one step in the future. □

Definition 43. Let $v \in \mathbb{EV}$ be non-empty and proper and $c, m \in \mathbb{N}$, such that $0 \leq c \leq \max\{0, v(1,1)\}$ and $m = \max_{s \in \mathbb{S}(v)}\{s(1,2)\} - 1$. The *space requirements* $SR(c,r)$ of v starting at c is

$$SR(c,v) = \left| \bigcup_{i=c}^{m} E(m, v:(i-c), v:(i-c)) \right| = \sum_{i=c}^{m} |E(i,v,v)| .$$

Def. 43 looks for the furthest position in the future asked for by the monitor and evaluates every n-step pair prior to that position. The n-step pairs represent the new monitor instances. It is a version of the concept described at the end of Sec. 2.3 formalized for evaluation vectors. The results of the next section are easier to discover and represent using this alternative formalization.

Now using Def. 43 we are able to connect the operational semantics of Sec. 2 with the abstraction found in this section.

Theorem 44. *Let $M \in \mathbb{M}$ such that $qt = QT(D(M)) \in \mathbb{QV}$ and $v = \mathbb{QP}(qt)$. Then for all $n, p, s, s' \in \mathbb{N}$, $ms \in \{\top, \bot\}^{\omega}$, such that $T(M) \multimap_{p,ms,n} s$, we have $s \leq SR(0,v)$.*

Proof (sketch). See (Cerna et al., 2016a). □

Example 11. By Lem. 42 and the table of Ex. 10, we can calculate $SR(0,v)$, where v is the vector from Ex. 10, as

$$SR(0,v) = \left| \bigcup_{i=0}^{6} E(6, v:i, v:i) \right| = \sum_{i=0}^{6} |E(i,v,v)| = \sum_{i=0}^{6} \sum_{j=0}^{2} |E(i,v,v)|^{j+1}$$

where v is the vector of Ex. 10. $SR(0,v)$ is the sum of all the columns of the matrix which is 159. Previous work (Cerna et al., 2016c) determines 1088 as the approximate bound for the space requirements of the vector. Thus, our new results provide a much tighter bound. In the next section we construct an analytic expression for bounds of both uniform and standard evaluation vectors.

4. Analysis of Uniform, Standard, and Inverse-standard Evaluation Vectors

The work outlined in this section is, for the most part, described in (Cerna et al., 2016a). We will repeat these results in this section being that they are necessary for the results we will derive in the following section. The new analysis added in this work is of inverse-standard evaluation vectors. This analysis is important for our analysis of both Top-free and general evaluation vectors. For any of the larger proofs please see (Cerna et al., 2016a).

4.1. Uniform Standard Evaluation Vectors

In this section we focus on the space complexity of uniform standard evaluation vectors. A precise bound is provided.

Theorem 45. *Let $v \in \mathbb{U}\$$ be a non-empty and proper. Then $|e(v(1,1),v,v)| = |v|$ and $|\overline{e}(v(1,1),v,v)| = 1$.*

Proof. See (Cerna et al., 2016a). □

Corollary 46. *Let $v \in \mathbb{U}\$$ be non-empty and proper. Then for $1 \leq i \leq |v|$,*

$$|e(v(1,1),v,v)|^i = 1.$$

Theorem 47. *Let $v \in \mathbb{U}\$$ be a non-empty and proper, and $i, m \in \mathbb{N}$, such that $1 \leq i \leq |v|$ and $v(1,1) \leq m < v(1,2)$. Then $|E(m,v,v)|^i = ((m - v(1,1)) + 1)^{i-1}$.*

Proof. See (Cerna et al., 2016a). This proof essentially proceeds by a complete induction on the pairs (m,i). Our induction hypothesis comes in three parts, namely, that the theorem holds for all (k,l) such that $k \leq m$ and $(l \leq i)$, it holds for all $(k, i+1)$ such that $k \leq m$ and, it holds for all $(m+1,l)$ such that $l \leq m$. Using these assumptions, we show that the theorem holds for $(m+1, i+1)$. In the end, we get a sum of the three parts which results in the sum of the binomial theorem. □

Thm. 47 can be used to derive the following space requirements for uniform standard evaluation vectors.

Corollary 48. *Let $v \in \mathbb{U}\$$ be a non-empty and proper, and $c \in \mathbb{N}$, such that $0 \leq c \leq v(1,1)$. Then*

$$SR(c,v) = (v(1,1) - c) + \sum_{i=1}^{m'} \sum_{j=0}^{|v|-1} i^j = ((v(1,1) + |v|) - c) + \left(\sum_{i=2}^{m'} \frac{(1 - i^{|v|})}{1 - i} \right)$$

where $m' = v(1,2) - v(1,1)$.

Proof. See (Cerna et al., 2016a). □

A direct consequence of Cor. 48 is that a partial evaluation function of a vector $v \in \mathbb{U}\$$ can be defined. This will be useful for the next section.

Definition 49. Let $v \in \mathbb{S}\mathbb{E}$ be a non-empty and proper, and $a, b \in \mathbb{N}$ such that $v(1,1) \leq a \leq b < v(1,2)$. We define the *partial evaluation of v* $P(a,b,v)$ as follows:

$$P(a,b,v) = \sum_{i=a}^{b} \sum_{j=0}^{|v|-1} ((i - v(1,1)) + 1)^j$$

Essentially Def. 49 allows us to choose the interval over which we evaluate the evaluation vector. Notice that we use a vector in $\mathbb{S}\mathbb{E}$ rather than $\mathbb{U}\$$.

4.2. Standard Evaluation Vectors

The evaluation pattern is slightly different for uniform standard and standard evaluation vectors. This difference is highlighted by the following lemma.

Lemma 50. *Let $v, h, g \in \mathbb{SE}$ be non-empty and proper and $l \in \mathbb{N}$ such that $0 \leq l \leq |\mathbb{S}(v)|$, h is (v,l)-perfect, and $v = h \otimes g$. Then for all $i, k \in \mathbb{N}$ such that $g(1,2) \leq k$ and $1 \leq i \leq |h|$,*

$$|E(k,v,v)|^i = |E(k,h,h)|^i = 0.$$

Essentially Lem. 50 states that, for every vector $w \in E(k,v,v)$, $i \leq ss(w,1)$, i.e. $\mathbf{h}_w = v(k)$ such that $i < k$. However, more importantly the perfect vectors of v evaluate like v. A consequence of Lem. 50 is that the functions $\overline{e}(\cdot,\cdot,\cdot)$ and $\overline{E}(\cdot,\cdot,\cdot)$ need to be considered for the analysis of standard evaluation vectors, see Cor. 38.

Lemma 51. *Let $v, h \in \mathbb{SE}$ be non-empty and proper, and $l \in \mathbb{N}$ such that $0 \leq l \leq |\mathbb{S}(v)|$ and h is (v,l)-perfect. Then $|\overline{E}(h(|h|,2),h,h)| = ((h(|h|,2) - h(1,1)) + 1)^{|h|}$.*

Proof. Essentially, $\overline{E}(h(|h|,2),h,h)$ is the entire degenerate set of h. The size of the degenerate set can be found as follows: consider h as the set of all words of length $|h|$ with an alphabet of size $((h(|h|,2) - v(1,1)) + 1)$. Members of the set are vectors $w \in SC_h$ such that $ss(w,1) = |h|$, subsets are vectors $w \in SC_h$ such that $ss(w,1) \leq |h|$. □

The above result only applies to the sections of the standard evaluation vector. If we were to apply them inductively the result is the following lemma:

Lemma 52. *Let $v, h, g \in \mathbb{SE}$ be non-empty and proper, and $l \in \mathbb{N}$, such that $0 \leq l \leq |\mathbb{S}(v)|$, h is (v,l)-perfect, and $v = h \otimes g$. Then $|\overline{E}(v(|v|,2),v,v)| = |\overline{E}(h(|h|,2),h,h)| \cdot |\overline{E}(v(|v|,2),g,g)|$.*

Proof. Essentially, this recursive property results from the construction of the degenerate set. □

Corollary 53. *Let $v \in \mathbb{SE}$ be non-empty and proper. Then*

$$|\overline{E}(v(|v|,2),v,v)| = \prod_{h \in \mathbb{S}(v)} ((h(1,2) - h(1,1)) + 1)^{|h|}.$$

Now we inductively apply our results for uniform standard evaluation vectors, Cor. 48, to the individual sections of a standard vector. This results in the following space bounds for standard evaluation vectors.

Theorem 54. *Let $v, h, g, w \in \mathbb{SE}$ be non-empty and proper, $l \in \mathbb{N}$, $s \in \mathbb{S}(v)$, such that $0 \leq l < |\mathbb{S}(v)|$, h is (v,l)-perfect, w is (v,l)-perfect, $v = h \otimes g$, and $w = h \otimes s$. Then for all $k, q \in \mathbb{N}$, such that $h(|h|,2) \leq k < w(|w|,2)$, $|h| < q < |v|$ the following holds:*

$$|E(k,v,v)|^q = |\overline{E}(h(1,2),h,h)| \cdot |E(k,g,g)|^{q-|l|} = \prod_{r \in \mathbb{S}(h)} ((r(1,2) - r(1,1)) + 1)^{|r|} \cdot ((k - s_1(1,1)) + 1)^{q-(|l|+1)}$$

Proof. See (Cerna et al., 2016a). □

Theorem 55. *Let* $v, h_1 \cdots, h_{|\mathbb{S}(v)|}, g_1 \cdots, g_{|\mathbb{S}(v)|} \in \mathbb{SE}$ *be non-empty and proper,* $c, i \in \mathbb{N}$*, such that* $0 \leq c \leq v(1,1)$*,* $1 \leq i \leq |\mathbb{S}(v)|$*,* h_i *is* (v,i)*-perfect, and* $v = h_i \otimes g_i$*. Then*

$$SR(c,v) = (v(1,1) - c) + P(v(1,1), v(1,2) - 1, v) + \sum_{i=1}^{|\mathbb{S}(v)|-1} \overline{E}(h_i(i,2), h_i, h_i) \cdot P(h_i(1,2), g_i(1,2) - 1, g_i)$$

Proof. This is a result of adding up all the possibilities of Thm. 54. □

Example 12. Using the standard evaluation vector of Ex. 10,

$$v = [(0,5,0), (0,6,0), (0,7,0)]$$

we get the following value for the runtime representation size when computing $SR(0,v)$.

$$SR(0,v) = P(0,4,v) + \sum_{i=1}^{2} \overline{E}(h_i(i,2), h_i, h_i) \cdot P(h_i(1,2), g_i(1,2) - 1, g_i) =$$

$$\sum_{i=0}^{4}\sum_{j=0}^{2}(i+1)^j + \overline{E}(5, [(0,5,0)], [(0,5,0)]) \cdot P(5,5,[(0,6,0),(0,7,0)]) +$$

$$\overline{E}(6, [(0,5,0),(0,6,0)], [(0,5,0),(0,6,0)]) \cdot P(6,6,[(0,7,0)]) =$$

$$3 + 7 + 13 + 21 + 31 + 6 * \left(\sum_{i=5}^{5}\sum_{j=0}^{1}(i+1)^j\right) + 42 * \left(\sum_{i=6}^{6}\sum_{j=0}^{0}(i+1)^j\right) =$$

$$75 + 42 + 42 = 159$$

Notice that the innermost sum of Thm. 55 behaves like a uniform standard evaluation vector evaluation, even though it contains multiple sections. This occurs because as long as an interval hasn't completely evaluated it behaves exactly the same as larger intervals. Unexpectedly, a naive computation is a good approximation of the space requirements of standard evaluation vectors, even outperforming the more complex analysis found in (Cerna et al., 2016c). We will show in later sections that this a naive computation is a good approximation for any evaluation vector.

Theorem 56. *Let* $v \in \mathbb{SE}$ *be non-empty and proper, and* $c \in \mathbb{N}$ *such that* $0 \leq c \leq v(1,1)$*. Then*

$$SR(c,v) \leq \prod_{i=1}^{j} I_i^{|s_i|} = O\left(I_j^{|v|}\right)$$

Proof. This is essentially replacing the partial evaluation with the degenerate set, obviously a worst case scenario. □

This result improves the $O\left(I_j^{2\cdot|v|}\right)$ space complexity bound presented in (Cerna et al., 2016c).

4.3. Analysis of Inverse-Standard Evaluation Vectors

In the previous section we provided a precise bound for the memory requirements of standard evaluation vectors. In this section we consider the case when the upper bounds of the sections are in the inverse order of the order found in standard evaluation vector, i.e. that of *inverse-standard evaluation vectors*. Most evaluation vectors are neither standard evaluation vectors nor inverse-standard evaluation vectors, However we will show that every top-free evaluation vector has space requirements between the space requirements of two vectors, one being standard and the other being inverse-standard, of which are built from the sections of the given top-free evaluation vector. We start with a theorem very similar to Thm. 47.

Theorem 57. *Let $v, s, w \in \mathbb{IS}$ be non-empty and proper, $h, g \in \mathbb{IS}$ be proper, $l, m \in \mathbb{N}$ such that $1 < l \leq \mathbb{S}(v)$ and $s(1,1) \leq m < s(1,2)$. Let h be $(v, l-1)$-perfect such that $v = h \otimes s \otimes g$, and $w = f(l, s)$ where f is defined as follows:*

$$f(n+1, s) = \begin{cases} s \otimes f(n+1, s) & n+1 > 0 \\ s & n+1 = 0 \end{cases}.$$

Then $|E(m, v, v)|^q = |E(m, w, w)|^q$ where $1 \leq q \leq |w|$.

Proof. We proceed by induction on l. If $l = 1$, Then g and h are empty vectors and $v = s = w$ and thus the equality holds. Let us assume the theorem holds for all d such that $0 \leq d \leq l < |v|$ and show that it hold for $l + 1$. For this induction to work we must assume that g is non-empty. Also, we distinguish between the s,w, the vectors of the induction hypothesis, and s',w' the vectors for the induction step. We assume that m was chosen such that $m < s'(1,2)$. Let $r = w \otimes s'$ and $w' = f(l+1, s')$. We know by the induction hypothesis that $|E(m, v, v)|^q = |E(m, w, w)|^q$. Let us assume that $q = |w|$. By Lem. 37, $|E(m, v, v)|^{q+q'} = |\overline{E}(m, h, h)| \cdot |E(m, s \otimes g, s \otimes g)|^{q'}$ for $1 \leq q' \leq |s \otimes g|$. By Thm. 47, $|E(m, s', s')|^{q'} = |E(m, s', s')|^{q'} = |E(m, s', s')|^{q'}$. It is clear from Def. 29, that $|\overline{E}(m, h, h)| = |\overline{E}(m, w, w)| = |\overline{E}(m, w^*, w^*)|$, where $w^* = f(l, s')$. Putting everything together we get

$$|E(m, v, v)|^{q+q'} = |\overline{E}(m, h, h)| \cdot |E(m, s', s')|^{q'} =$$

$$|\overline{E}(m, w^*, w^*)| \cdot |E(m, s', s')|^{q'} = E(m, r, r)|^{q+q'}$$

□

Lemma 58. *Let $v, s \in \mathbb{IS}$ be non-empty and proper,$h, g \in \mathbb{IS}$ be proper, $l, m \in \mathbb{N}$ such that $1 < l \leq \mathbb{S}(v)$ and $s(1,1) \leq m < s(1,2)$, and h be $(v, l-1)$-perfect such that $v = h \otimes s \otimes g$. Then*

$$|E(m, v, v)|^q = 0$$

where $|h| < q \leq |v|$.

Proof. This follows directly from **E5** in Def. 29. □

Corollary 59. *Let $v, s \in \mathbb{IS}$ be non-empty and proper, $m \in \mathbb{N}$ such that $s(1,1) \leq m < s(1,2)$, and s be $(v,1)$-perfect. Then*

$$|E(m,v,v)| = |E(m,s,s)| = \sum_{j=0}^{|s|-1} ((m - s(1,1)) + 1)^j$$

Proof. Put Lem. 58 and Thm. 47 together. □

What we have shown so far is that inverse-standard evaluation vectors behave like uniform standard evaluation vectors which shrink as the position at which they are evaluated increases.

Corollary 60. *Let $v, s, w \in \mathbb{IS}$ be non-empty and proper where s is $(v,1)$-perfect, $m, k \in \mathbb{N}$ such that $s(1,1) \leq m < s(1,2)$ and $1 \leq k \leq |\mathbb{S}(v)|$. Let w be (v,k)-perfect where $w(1,2) > m$ and for the $(v,k+1)$-perfect vector w', $w'(1,2) \leq m$. Then*

$$|E(m,v,v)| = \sum_{r=0}^{|w|-1} ((m - v(1,1)) + 1)^r$$

The following corollary is easier to write without the use of perfect vectors.

Corollary 61. *Let $v = s_1 \otimes \cdots \otimes s_j$ be a non-empty proper inverse-standard evaluation vector, and $c, q, k \in \mathbb{N}$ such that $0 \leq c < s_1(1,2)$, $q = \sum_{i=1}^{k} |s_i|$, and $k = \max\{i | s_i(1,2) < m\}$. Then*

$$SR(c,v) = (v(1,1) - c) + \sum_{m=s_1(1,1)}^{s_1(1,2)-1} \sum_{r=0}^{q(m)} ((m - s_1(1,1)) + 1)^r$$

where $q(m) = \sum_{i=1}^{k(m)} |s_i|$, and $k(m) = \max\{i | s_i(1,2) < m\}$.

Note that by Cor. 61 the space requirements of inverse-standard evaluation vectors are similar to those of uniform standard evaluation vectors. The major difference is the shrinking of the vector represented by $q(m)$ in Cor. 61. This observation leads us to the next result.

Theorem 62. *Let $v \in \mathbb{IS}$ and $w \in \mathbb{SE}$ be non-empty and proper such that $\mathbb{S}(v) = \mathbb{S}(w)$. Let $c \in \mathbb{N}$ be such that $0 \leq c \leq v(1,1)$. Then*

$$SR(c,v) \leq SR(c,w)$$

Proof. We proceed by induction on $|\mathbb{S}(v)|$. If v has a single section, i.e. $j = 1$, then $v = w$, and thus the theorem holds. For the induction hypothesis, we assume the theorem holds for v such that $|\mathbb{S}(v)| = j$ and show that the theorem holds for vectors v' and w' where $|\mathbb{S}(v)| \subset |\mathbb{S}(v')| = j + 1$ and $|\mathbb{S}(v')| = |\mathbb{S}(w')|$. Let $s \in \mathbb{US}$ be such that $v' = v \otimes s$. We can write $SR(c,v')$ as follows:

$$SR(c,v') = (v'(1,1) - c) + \sum_{k=1}^{I-1} \sum_{l=|v|+1}^{|v'|} k^{l-1} + SR(v(1,1), v)$$

where $I = ((s(1,2) - s(1,1)) + 1)$. We can write $SR(c, w')$ as

$$SR(c, w') = (w'(1,1) - c) + \sum_{k=1}^{I-1} \sum_{l=1}^{|w'|} k^{(l-1)} + (I_{j+1})^{|s|} \cdot SR(w'(1,1), w).$$

To prove the theorem we have to show that

$$\sum_{k=1}^{I-1} \sum_{l=|v|+1}^{|v'|} k^{l-1} \leq \sum_{k=1}^{I-1} \sum_{l=1}^{|w'|} k^{(l-1)} + (I_{j+1})^{|s|} \cdot SR(w'(1,1), w)$$

holds. We can derive the following inequality by subtracting the left side from both sides,

$$0 \leq \sum_{k=1}^{I-1} \sum_{l=1}^{|v|} k^{(l-1)} + ((I_{j+1})^{|s_j|} - 1) \cdot SR(w'(1,1), w),$$

and thus show that the theorem holds. Note that the right-hand side can easily be shown to be greater than zero. □

From Thm. 62, an even stronger result follows when we assume $j > 1$.

Corollary 63. *Let $v \in \mathbb{IS}$ and $w \in \mathbb{SE}$ be non-empty and proper such that $\mathbb{S}(v) = \mathbb{S}(w) > 1$. Let $c \in \mathbb{N}$ be such that $0 \leq c \leq v(1,1)$. Then $SR(c, v) < SR(c, w)$.*

5. Bounds for Top-free Evaluation Vectors

Our goal in this section is to show that even though a complete analysis of top-free evaluation vectors is difficult to carry out from the results presented so far in this paper, we can still show that the space requirements of any top-free evaluation vector is bounded from above and from below. The upper bound is provided by a standard evaluation vector and the lower bound by an inverse-standard evaluation vector. Thm. 56 shows that the current results are already better than previous work concerning standard evaluation vectors. In this section, we show that the results are better for almost any vector. Before proving these bounds, we need to define a few new concepts needed in the proofs.

Definition 64. Let V be a set of non-empty proper top-free evaluation vectors such that there exists some $a \in \mathbb{N}$ such that for all $v \in V$, $v(1,1) = a$. We define $V^{\otimes}$ inductively as follows:

$$V^{\otimes} = \left\{ \{v \otimes w\} \cup \{w \otimes v\} \,|\, v \in V \ \wedge \ w \in (V \setminus v)^{\otimes} \right\} \qquad \text{if } V \neq \emptyset$$
$$\emptyset^{\otimes} = \{[]\} \qquad \text{otherwise}$$

Definition 65. Let V be a set of non-empty proper top-free evaluation vectors such that there exists some $a \in \mathbb{N}$ such that for all $v \in V$, $v(1,1) = a$. We define *a standard evaluation vector relative to V*, $\mathbf{S}(V)$, as a vector $v \in V^{\otimes}$ such that, for all $w \in V^{\otimes}$, $SR(a, w) \leq SR(a, v)$. Also, *a inverse-standard evaluation vector relative to $V^{\otimes}$*, $\mathbf{I}(V)$, is a vector $v \in V^{\otimes}$ such that, for all $w \in V^{\otimes}$, $SR(a, v) \leq SR(a, w)$.

Definition 66. Let V be a set of non-empty proper top-free evaluation vectors such that there exists some $a \in \mathbb{N}$ such that for all $v \in V$, $v(1,1) = a$. We define *a standard evaluation vector of* $V^{\otimes}$, $s(V)$, as a vector $v \in V^{\otimes}$ such that, $v = v^1 \otimes \cdots \otimes v^{|V|}$ and $v^1(1,2) \leq \cdots \leq v^{|v|}(1,2)$. Also,we define *a inverse-standard evaluation vector of* $V^{\otimes}$, namely $i(V)$ as a vector $v \in V^{\otimes}$ such that, $v = v^1 \otimes \cdots \otimes v^{|V|}$ and $v^1(1,2) \geq \cdots \geq v^{|v|}(1,2)$.

Lemma 67. *Let V and V' be non-empty sets of non-empty proper uniform standard evaluation vectors such that there exists some $a \in \mathbb{N}$ such that for every $v \in V$ and $v' \in V'$ $v'(1,1) = v(1,1) = a$. Let $v^1, v^2 \in V^{\otimes}$ such that $SR(a, v^1) \leq SR(a, v^2)$. Then for all $w \in (V')^{\otimes}$, $SR(a, w \otimes v^1) \leq SR(a, w \otimes v^2)$ and $SR(a, v^1 \otimes w) \leq SR(a, v^2 \otimes w)$.*

Proof. Consider that $SR(a, v^1) \leq SR(a, v^2)$ implies $\overline{E}(m, v^1, v^1) \leq \overline{E}(m, v^2, v^2)$ for all m, i.e. if there is more in memory there is also more to remove. This lemma than becomes a simple application of Lem. 37 and Cor. 38. □

Theorem 68. *Let V be a non-empty set of non-empty proper uniform standard evaluation vectors such that there exists $a \in \mathbb{N}$ such that for every $v \in V$ such that $v(1,1) = a$. Then $\mathbf{S}(V) = s(V)$ and $\mathbf{I}(V) = i(V)$.*

Proof. We proceed by induction on the size of V. If $V = 1$, the theorem trivially holds being that there is only one vector in $V^{\otimes}$.

Let us assume the theorem holds for V such that $1 \leq |V| \leq j$. We show that it hold for V such that $|V| = j + 1$. We take an arbitrary vector $w \in V^{\otimes}$, $w = v_1 \otimes \cdots \otimes v_{j+1}$, where $v_i \in V$ for $1 \leq i \leq j + 1$. We break w into two parts $w^1 = v_1$ and $w^2 = v_2 \otimes \cdots \otimes v_{j+1}$. Notice that $|w^2| = j$ and thus, by the induction hypothesis $SR(a, w^2) \leq SR(c, s((V \setminus v_1)))$. By Lem. 67, we know that $SR(a, w) \leq SR(c, w^1 \otimes s((V \setminus v_1)))$. Now we have to consider two cases, either $v_1(1,2) \geq v_i(1,2)$ for $1 \leq i \leq j + 1$ or there exists some l such that $1 < l \leq j + 1$, such that $v_l(1,2) \geq v_i(1,2)$, for all i such that $1 \leq i \leq j + 1$.

If the second case holds, we break $s((V \setminus v_1))$ into two parts, $s_1 = s((V \setminus \{v_1, v_l\}))$ and $s_2 = v_l$. The vector $v_1 \otimes s_1$ has a length of j and thus by the induction hypothesis $SR(c, v_1 \otimes s_1) \leq SR(c, s((V \setminus v_l)))$. By Lem. 67, we know that $SR(c, v_1 \otimes s_1 \otimes s_2) \leq SR(c, s((V \setminus v_l)) \otimes s_2)$. We have shown that

$$s((V \setminus v_l)) \otimes s_2 = s(V) = \mathbf{S}(V)$$

If the first case holds, we break $s((V \setminus v_1))$ into two parts, $s_1 = s((V \setminus \{v_1, v_l\}))$ and $s_2 = v_l$. The vector $v_1 \otimes s_1$ has a length of j and thus by the induction hypothesis $SR(c, v_1 \otimes s_1) \leq SR(c, s((V \setminus v_l)))$. By Lem. 67, we know that $SR(c, v_1 \otimes s_1 \otimes s_2) \leq SR(c, s((V \setminus v_l)) \otimes s_2)$. However, $s((V \setminus v_l)) \otimes s_2 \neq s(V)$. Let $s((V \setminus v_l)) \otimes s_2 = g$. we can break g into two parts g^1 and g^2 where g^1 is the first section of $s((V \setminus v_l)) \otimes s_2$ and g^2 is the rest of $s((V \setminus v_l)) \otimes s_2$. We are now back to the previous case. We repeat the previous argument on g to prove the theorem for $\mathbf{S}(V)$ and $s(V)$. Proving the theorem for $\mathbf{I}(V)$ and $i(V)$ is the same except we use the opposite direction of Lem. 67. □

Theorem 69. *Let V be a non-empty set of non-empty proper uniform standard evaluation vectors such that there exists $a \in \mathbb{N}$ such that for every $v \in V$ we have $v(1,1) = a$. Let $w \in V^{\otimes}$. Then*

$$SR(a, i(V)) \leq SR(a, w) \leq SR(a, s(V))$$

Proof. Follows directly from Thm. 68. □

Theorem 70. *Let $v = s_1 \otimes \cdots \otimes s_j$ be a non-empty proper top-free evaluation. Then*

$$SR(a, i(V)) \leq SR(a, v) \leq SR(a, s(V))$$

where $V = \bigcup_{1 \leq i \leq j} \{s_i\}$.

Proof. Follows directly from Thm. 68. □

6. Analysis of Evaluation Vectors

A problem with Thm. 69 is that it only holds for top-free evaluation vectors which have a constraint on the lower bound. If we want to apply the above results to any evaluation vector, we need to figure out a method to transform the vectors into a top-free form. A naive shifting of the intervals in order to equalize the lower bounds does not preserve space requirements. For example consider the evaluation vector

$$[(5, 10, 0), (0, 2, 0)] \Rightarrow [(0, 5, 0), (0, 2, 0)] .$$

Prior to transformation, at position 3 there are 11 instances, after transformation there are 7 instances. If instead of $(0, 2, 0)$ we use $(0, 4, 0)$, the number of instances after transformation will be 14, but the number of instances prior to the transformation remains the same. This inconsistency illustrates that such a transformation is not a viable solution. However, another solution is to add in missing instances, which we will call finding the *negative* of an evaluation vector. This method is better in the sense that we are still computing the space requirements of the original vector, however, with additional instances added. Essentially, we will get a coarser solution.

Definition 71. Let v be a proper evaluation vector and v' be a proper top-free evaluation vector such that $|v| = |v'|$, and for all $1 \leq i \leq |v|$ we have $v(i, 2) = v'(i, 2)$ and $v'(i, 1) = \min_{1 \leq j \leq |v|} \{v(j, 1)\}$. The *negative of* v is a proper top-free evaluation vector w such that $|w| = |v'|$, and for all $1 \leq i \leq |v|$ we have $v'(i, 1) = w(i, 1)$ and $v(i, 1) = w(i, 2)$. We refer to v' as the *image of* v *and* w.

Because the negative w of a vector v, is top-free, we know, by Thm. 70, that there is some inverse-standard evaluation vector I, based on the structure of w, such that $SR(w(1, 1), I) \leq SR(w(1, 1), w)$. Also, by Thm. 70, there is a standard evaluation vector S such that the image v' of v and w, is bounded by it, i.e. $SR(v'(1, 1), v') \leq SR(v'(1, 1), S)$. Putting all this together, we can derive the following theorem.

Theorem 72. *Let v be a proper evaluation vector, $w = s_1 \otimes \cdots \otimes s_j$ be the negative of v, and $v' = s'_1 \otimes \cdots \otimes s'_k$ the image of v and w. Let $V = \bigcup_{1 \leq i \leq j} \{s_i\}$ and $V' = \bigcup_{1 \leq i \leq k} \{s'_i\}$. Then*

$$SR(v'(1, 1), v) \leq SR(v'(1, 1), s((V'))) - SR(v'(1, 1), i(V)).$$

Proof. We know the transformation of v is adding extra instances to the space requirements of v. The minimal number of instances possibly added is $SR(v'(1, 1), i(V))$. Thus, subtracting This amount from an over approximating upper bound $SR(v'(1, 1), s((V')))$ gives us a more accurate bound. It is quite obvious that $SR(v'(1, 1), v) \leq SR(v'(1, 1), s((V')))$ holds. □

Thus Thm. 72 provides a method to bound any evaluation vector which, however, is still not as accurate as Thm. 69 is for top-free evaluation vectors. Nevertheless, it is again better than the results of previous work (Cerna et al., 2016c) and the application of Thm. 69 alone.

7. An Algorithm for Computing Space Requirements

In this section we will first discuss some experimental results which motivated our investigations into constructing an algorithm for computing the space requirements of general formula. Then we will introduce annotated quantifier trees, which are quantifier tree with additional information needed for computing the runtime representation size. We then introduce an algorithm for computing the runtime representation size and end the section with experimental results.

7.1. Motivating Experiments

Both Thm. 69 & 70 provide a precise upper and lower bound for every top-free evaluation vector constructed from a set of uniform standard evaluation vectors. Though, we do not know how accurate these bounds are for a particular top-free evaluation vector. What is obvious from simple experiments is that the gap between the lower and upper bound can be quite large. Take for example the following set of uniform standard evaluation vectors:

$$V = \left\{ \begin{array}{lll} [(0,1,0)] & [(0,2,0)] & [(0,3,0)] \\ [(0,4,0)] & [(0,5,0)] & [(0,6,0)] \end{array} \right\}$$

We can easily compute that

$$s(V) = [(0,1,0),(0,2,0),(0,3,0),(0,4,0),(0,5,0),(0,6,0)]$$

and

$$i(V) = [(0,6,0),(0,5,0),(0,4,0),(0,3,0),(0,2,0),(0,1,0)]\,.$$

The space requirements for the two vectors are as follows: $SR(0, s(V)) = 2252$ and $SR(0, i(V)) = 105$. The gap between the two values is quite large in that $SR(0, i(V))$ is roughly a mere 5% of $SR(0, s(V))$. For larger sets the gap is even larger. However, this simple analysis does not address the topological structure of the space requirements of the other $n! - 2$ vectors in V. We would like to know if all lie on a line between 2252 and 105, or if there is some internal structure which can be used to get better/more precise results. In turns out that there is a regular recursive pattern ordering the space complexity of the constructable vectors. We do not derive a formal description of this pattern, but rather highlight its existence.

We have implemented our evaluation procedure (Def. 29, Def. 43) and ran simulations for all vectors constructable from the set $V^i = \{[(0,i,0)]\} \cup V^{i-1}$ where $V^1 = \{[(0,1,0)]\}$. Specifically, we worked with the sets V^i for $i \in [1,2,\cdots,7]$. The vectors are depicted in Fig. 8 & 9. Concerning the sets $V^1, \cdots V^5$, there is some structure visible but it is not very substantial. However, for V^6 and V^7 it is quite obvious that certain vectors will always mark jumps in the space requirements. Notice that position 5 and 6 from Fig. 8 have pretty much the same shape as position 3 and 4 from Fig. 8. Essentially, the location of the relatively largest interval is an important indicator of the runtime representation size.

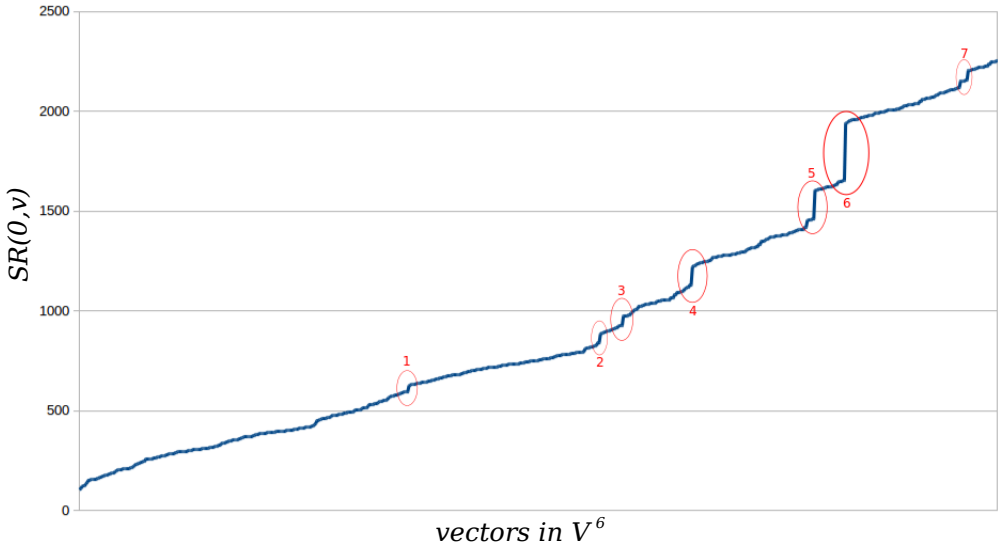

Fig. 8. Space requirement results for vectors in V^6. Numbers mark jumps in number of instances.

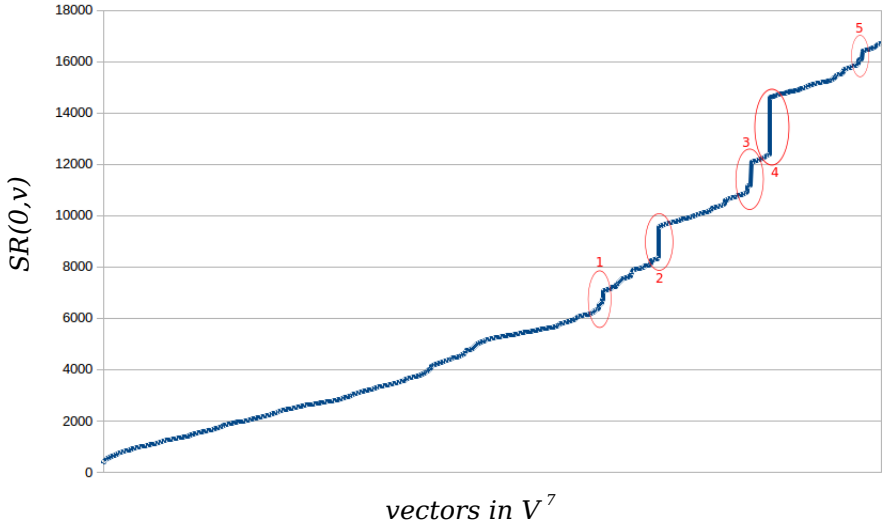

Fig. 9. Space requirement results for vectors in V^7. Numbers mark jumps in number of instances.

The vectors corresponding to position 5 from Fig. 8 are

$$[(0,1,0),(0,2,0),(0,3,0),(0,6,0),(0,4,0),(0,5,0)]$$

$$[(0,4,0),(0,3,0),(0,2,0),(0,1,0),(0,6,0),(0,5,0)]\,,$$

where the first vector is the bottom of the jump and the second is the top. Comparing to the position 3 from Fig. 9, the vectors are very similar,

$$[(0,1,0),(0,2,0),(0,3,0),(0,4,0),(0,7,0),(0,5,0),(0,6,0)]$$

$$[(0,5,0),(0,4,0),(0,3,0),(0,2,0),(0,1,0),(0,7,0),(0,6,0)]\,.$$

Essentially, the jump seems to occur when specific parts are saturated, i.e. in standard evaluation vector form, and the only way to force more instances in memory is to move to an inverse-standard evaluation vector form with a larger part. This pattern can be seen in the other jumps as well. This indicates that the permutation of the upper bounds plays a significant role in the computation of the runtime representation size. This information can be used to construct a general procedure for a precise computation of the runtime representation size.

7.2. Annotated Quantifier Tree

We now introduce the notion of annotated quantifier tree.

Definition 73 (Size Annotation)**.** We define the size annotation $A : \mathbb{QT} \to^{\text{part.}} \mathbb{Z} \cup \{\infty\}$ (whose domain is the set of quantifier trees resulting from the dominating form of a monitor) recursively as follows:

$$
\begin{aligned}
A((V, \infty, B, qt)) &= 0 \\
A((V, c_1, x + c_2, qt)) &= \begin{cases} \max\{c_1, c_2\}, & \text{if } \forall q \in qt.\ A(q) \leq 0 \\ \infty, & \text{otherwise} \end{cases} \\
A((V, x + c_1, c_2, qt)) &= A((V, x + c_1, x + c_2, qt)) = A((V, c_1, c_2, qt)) \\
&= \max\{c_1, c_2, \max_{q \in qt}\{A(q)\}\} \\
A((V, x + c_1, \infty, qt)) &= A((V, c_1, \infty, qt)) = \infty
\end{aligned}
$$

Notice that the annotation takes care of the cases when the evaluation of a formula requires an infinite amount of memory. There are three such cases, the most complex one being $(V, c_1, x + c_2, qt)$: here the amount of memory needed increases over time if qt requires a positive amount of memory, because every time we generate a new monitor instance the interval increases. This occurs while we are still evaluating the previous instances. These two factors together result in an unbounded number of instances.

The point of this annotation is to indicate at what position a monitor instance's runtime representation will have size zero. Assume we are dealing with monitor instance $x = m$, when this instance is evaluated at position $A + n$ for $m \leq n$, the runtime representation is of size zero. When $m \geq n$ the runtime representation will have a size greater than zero. When $m > A + n$, the monitor instance cannot be evaluated at all and we end up with a runtime representation with size one. Our algorithm only considers monitor instances such that $n < m \leq A + n$.

Definition 74 (Annotated Quantifier Trees)**.** An *annotated quantifier tree* is inductively defined to be either $\emptyset$ or a tuple of the form (a, b_1, b_2, Q) where $a \in \mathbb{Z} \cup \{\infty\}$, $b_1, b_2 \in \mathbb{Z} \cup \{\infty\}$ and Q is a set of annotated quantifier trees. Let $\mathbb{AQT}$ be the set of all annotated quantifier trees.

Definition 75 (Annotated Quantifier Tree Transformation)**.** We define $AQT : \mathbb{QT} \to^{\text{part.}} \mathbb{AQT}$ (whose domain is the set of quantifier trees where only the monitor variable x

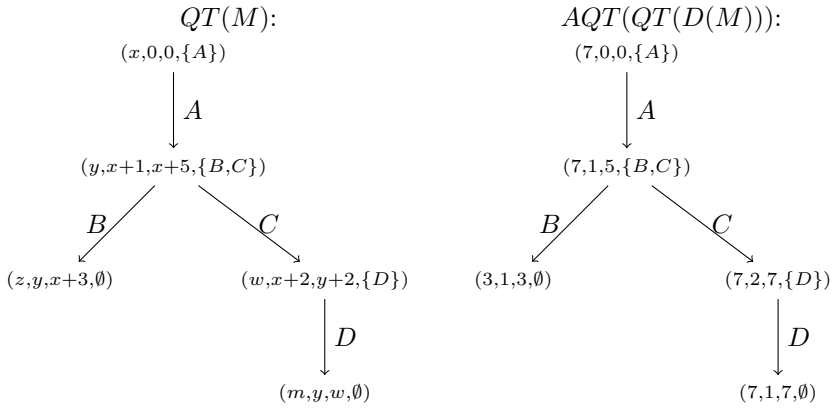

Fig. 10. (Annotated) Quantifier Trees

occurs in bounds) recursively as follows:

$$
\begin{aligned}
AQT((V, x + c_1, x + c_2, qt)) &= (A((V, x + c_1, x + c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, c_2, qt)) &= (A((V, c_1, c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, x + c_1, c_2, qt)) &= (A((V, x + c_1, c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, x + c_1, \infty, qt)) &= (A((V, x + c_1, \infty, qt)), c_1, \infty, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, \infty, qt)) &= (A((V, c_1, \infty, qt)), c_1, \infty, \cup_{q \in qt} AQT(q)) \\
AQT((V, \infty, x + c_1, qt)) &= (A((V, \infty, x + c_1, qt)), \infty, c_1, \cup_{q \in qt} AQT(q)) \\
AQT((V, \infty, c_1, qt)) &= (A((V, \infty, c_1, qt)), \infty, c_1, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, x + c_2, qt)) &= (A((V, c_1, x + c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q)) \\
AQT((V, c_1, c_2, qt)) &= (A((V, c_1, c_2, qt)), c_1, c_2, \cup_{q \in qt} AQT(q))
\end{aligned}
$$

Notice that if any subtree of an annotated quantifier tree requires infinite memory, then the uppermost node of the tree, i.e. the root, will have an annotation of ∞. Also, if the monitor represented by the annotated quantifier tree is completely backwards looking, then the annotation at the root will be 0. Thus, in these two cases no further computation is necessary to compute the space complexity of the monitor. Also note that we drop the variable from the bounds. This means that the bounds c_1 and $x + c_1$ are treated the same. This is not problematic being that our algorithm only considers the case when x maps to zero. To deal with cases $x \geq 0$, we consider the monitor instance created at $x = 0$ at various future positions.

Example 13. Let us consider the monitor specification M from Ex. 2. Then $QT(M)$ and $AQT(QT(D(M)))$ are as depicted in Figure 10.

7.3. Algorithm for Computing the Runtime Representation Size

The idea behind our algorithm is that the maximum number of instances for a specification with finite intervals occurs when we reach the furthest future position

Algorithm 1 Space Requirements of an Annotated Quantifier Tree

```
1: function SR(aqt)                          ▷ aqt is an annotated quantifier tree (A, a, b, qt')
2:     if A = ∞ then
3:         return ∞
4:     else
5:         return Σ_{i=0}^{A-1} SR(aqt, i)
6:     end if
7: end function
8:
9: function SR(aqt,i)                        ▷ aqt is an annotated quantifier tree (A, a, b, Q), i < A
10:    cil ← 1 + min {i, b} − a
11:    if cil ≤ 0 & b ≥ a then
12:        return 1
13:    else if cil ≤ 0 then
14:        return 0
15:    else
16:        if i ≥ b then
17:            inst ← 0
18:        else
19:            inst ← 1
20:        end if
21:        for all aqt' = (A', a', b', Q') ∈ Q do
22:            if i < A' then
23:                inst ← inst + cil · SR(aqt', i)
24:            end if
25:        end for
26:    end if
27:    return inst
28: end function
```

required by the first instance generated. This idea was also used in (Cerna et al., 2016c), though in a less general abstraction. After reaching this furthest future position there will be an instance corresponding to each of the previous positions reached. Thus, to calculate the runtime representation size all we need to do is calculate how the first instance behaves at each position prior to the furthest future position. This is exactly what the algorithm in Fig. 1 does. The annotations of the quantifier tree nodes are used to define the furthest future position, as well as, to check for infinite runtime representation size. Also, we provide a theorem similer to Thm. 44 for annotated quantifier trees (Thm. 76). In a nutshell, our analysis now proceeds as follows:

(1) We compute from a monitor $M \in \mathbb{M}$ the *dominating monitor* $M' = D(M) \in \mathbb{M}$ whose space requirements on the one hand bound the requirements of M and on the other hand can be determined exactly by the subsequent analysis.
(2) We translate $M' \in \mathbb{M}$ into a *quantifier tree* $qt = QT(M')$ which contains the essential information required for the analysis.
(3) We translate qt into an *annotated quantifier tree* $aqt = AQT(qt)$ which labels every node with the maximum interval bound of the corresponding subtree.
(4) Finally, we compute $SR(aqt) \in \mathbb{N}$ by application of Algorithm 1.

Theorem 76. *Let $M \in \mathbb{M}$ and $aqt = AQT(QT(D(M)))$. Then for all $n, p, S \in \mathbb{N}$ and $s \in \{\top, \bot\}^\omega$ such that $T(M) \multimap_{p,s,n} S$, we have $S \leq SR(aqt)$.*

Proof (sketch). Ignoring the special cases that the algorithm considers, i.e. the annotation of infinite memory, or subtrees which evaluate instantly, the heart of the algorithm is the observation that the quantifiers in dominating monitors can be treated the same independently of their position in the formula. This is not the case for non-dominating monitors because there is dependence between the intervals.

A second important observation is that the evaluation of the runtime monitors is independent of the position of the stream. Thus, we can take a single monitor instance and evaluate it at different positions to understand how all instances of the monitor will evaluate.

Going back to the first observation and Def. 1 & 2, we can consider the evaluation of a monitor M with a single quantifier whose interval is $[x + a, x + b]$, where $a \leq b$ and $a, b \in \mathbb{N}$. For $n \geq b$ it is easy to compute that $T(M) \multimap_{p,s,n} (b - a) + 1$. However, at positions $a \leq n < b$, $T(M) \multimap_{p,s,n} (n - a) + 1$. These results can already be found in (Cerna et al., 2016c). Since the instance production of quantifiers is independent of their location in a formula, we can use these two basic results to compute the number of instances of the quantified formula produced. An elementary but tedious inductive argument completes the proof: Take a monitor with m quantifiers and construct a new monitor such that the monitor's formula has an additional quantifier added on top, check that the algorithm holds for all relevant configurations of adding the new quantifier. □

Theorem 77. *Let $aqt = (A, b_1, b_2, aqt') \in \mathbb{AQT}$. Then $SR(aqt) = O(A^n)$ where $n = d(aqt)$ is the quantifier depth of aqt inductively defined by $d(\emptyset) = 0$ and $d(a, b_1, b_2, Q) = 1 + \max_{aqt \in Q} d(aqt)$.*

Proof (sketch). It is well known that $\sum_{i=0}^{A-1} i^n = O(A^n)$. If every quantifier in aqt has an interval $[0, A]$, then this summation accurately represents the computation of this algorithm: the outer $SR(\cdot)$ function represents the summation and the inner function $SR(\cdot, \cdot)$ computes the n^{th} degree polynomial. □

This result improves the $O\left(A^{2n}\right)$ space complexity bound presented in Cerna et al. (2016c).

7.4. Experimental Results

We have experimentally validated the predictions of our analysis for the following monitors where (1a) and (2a) represent the dominating forms of the monitors (1b) and (2b), respectively:

$$\forall_{0 \leq x} : \forall_{y \in [x, x+80]} : \forall_{z \in [x, x+80]} : @z \quad (1a) \qquad \forall_{0 \leq x} : \forall_{y \in [x, x+40]} : \forall_{z \in [x, x+80]} : @z \quad (2a)$$

$$\forall_{0 \leq x} : \forall_{y \in [x, x+80]} : \forall_{z \in [x, y]} : @z \quad (1b) \qquad \forall_{0 \leq x} : \forall_{y \in [x, x+40]} : \forall_{z \in [x, y+40]} : @z \quad (2b)$$

The diagram in Figure 11 displays on the vertical axis the number of formula instances reported by the LogicGuard runtime system for corresponding monitors in the real specification language; the horizontal axis displays the number of messages observed so far on the stream. The monitors are defined such that the body of the innermost quantifier always evaluates to true and thus always the full quantifier range is monitored and the

worst-case space complexity is exhibited. One should note that the runtime system reports the number of formula instances while our analysis determines a measure for the size of the monitor's runtime representation (which is difficult to determine in the real system); however, for monitors with less than three nested quantifiers, such as the ones given above, the results coincide (the z-quantifier does not store any instances, since its body is propositional; the y quantifier contains instances of size 1; the runtime system reports the number of these instances which is identical to the total size of these instances determined by our analysis).

As expected, we can observe that the number of instances eventually reaches, after the start-up phase, an upper bound. For the dominating monitors 1a and 2a, the predictions 1 (3320) and 2 (2459) reported by the analysis accurately match the observations. As expected, however, these predictions overestimate the number of instances observed for the non-dominating monitors 1b (160) and 2b (1659), from which the dominating monitors were derived. Interestingly, the over approximation for monitor 2b (by a factor of 1.5) is much less than for monitor 1b (by a factor of 21). It seems that our analysis is better at predicting the number of instances for certain quantifier configurations. This would imply that quantifier configurations which we cannot predict well (i.e., where the difference between the actual space requirements and that of the dominating form is large) may have better performance in real-world scenarios. This is a topic that we are going to investigate in future work.

We have also tested our algorithm with the following more realistic monitoring scenario which is based on the full language sketched in Section 2:

```
type int; type message; stream<int> IP;
stream<int> S = stream<IP> x satisfying @x>=0 :
      value[seq,@x,plus]<IP> y with x < _ <=# x+10000: @y;
monitor<S> M = monitor<S> x :
      forall<S> y with x < _ <=# x+15000:
        exists<S> z with y < _ <=# y+4000: IsEven(#z);
```

Though our algorithm is defined for an abstract of the core language we can translate any specification of the full language into concepts of the core language such that the resulting monitor in the core language has space requirements at least as big as the original monitor. As one can imagine, quantifiers such as `forall<S>` and `exists<S>` can be directly translated into core language quantifiers. More complex quantifiers require extra User input concerning the number of messages per millisecond and expect waiting time for unbounded loops.

Our realistic example creates an internal stream of values and constructs a monitor for the internal stream. The monitor checks whether within an interval of four seconds there is a message with an even arrival time. we check all 4 second intervals for 15 seconds following the creation of an instance. Whenever the 4 second interval does not contain a message with an even arrival time the monitor reports a violation. Unlike the artificial examples which do not have violations, The space usage of our realistic example does not have a smooth curve over the number of received messages. Notice that the peaks of the curve never go above the gray line which is the predication of our algorithm (see Fig. 12).

8. Conclusion

We have provided a new abstraction of the operational semantics of the LogicGuard core language (LogicGuard II, 2015) that is more precise than what has been presented

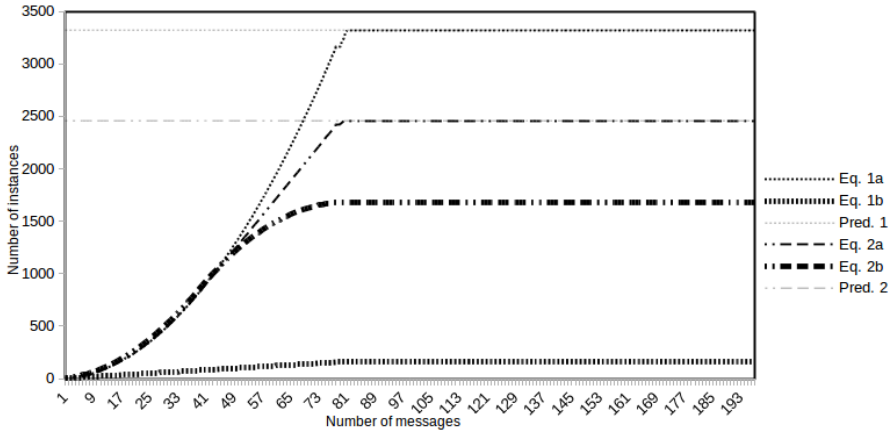

Fig. 11. Artificial experimental results versus predictions

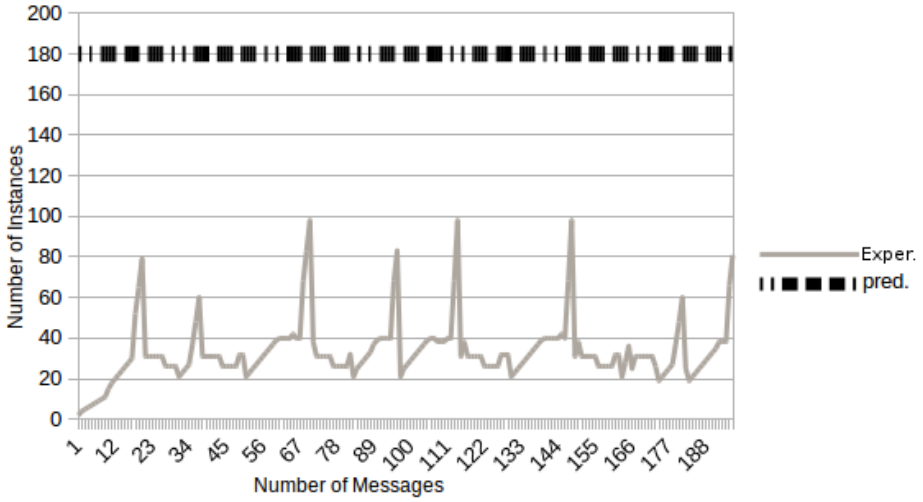

Fig. 12. Realistic experimental results versus predictions

in previous work (Cerna et al., 2016c); we also have extended the results so far attained using this abstraction (Cerna et al., 2016a). We have focused on an important class of monitor specifications, that is those which can be represented as evaluation vectors. This class consists of monitors whose quantifiers are future looking and are arbitrarily nested, but in the matrix of each quantifier, there is at most one non-nested quantifier. Our result concerning the subclass of uniform standard evaluation vectors can be used to derive precise bounds for the subclass of standard evaluation vectors and inverse-standard evaluation vectors.

These two subclasses were chosen because they represent upper and lower bounds, in terms of space requirements, for top-free evaluation vectors. We have shown that, given a set of top-free evaluation vectors, it is possible to construct a standard evaluation vector

which is an upper bound of the space requirements of any member of the set, and an inverse-standard evaluation vector which is an lower bound of the space requirements of any member of the set. These results can be extended further to provide bounds for any evaluation vector by transforming the evaluation into a top-free evaluation vector. The top-free evaluation vector which results from this transformation bounds the space requirement of the initial evaluation vector from above. Though, this transformation is quite coarse. To get a better bound, we go on to show that one can remove some instances and still bound the initial evaluation vector from above.

Though these results are quite good, we wanted to know if there is a way to precisely compute the bounds of any dominating quantifier tree. After a preliminary investigation into underlying patterns in the results for evaluation vectors we realized that one can algorithmically capture the runtime representation size. An implementation of this algorithm can be found in the LogicGuard software. We have provided an analysis of this algorithm to show how it precisely finds the size of the runtime representation of a dominating quantifier tree and bounds the runtime representation of a quantifier tree. We also have provided a real world test example written in the full specification language.

Concerning future work, there is still one facet of the runtime representation we have not considered, that is the number of accesses to the value of a position per time unit, which is essentially the time complexity of evaluating the specification. We expect this measure to be similar to the work presented here. Also, we plan to study the relationship between the dominating formula and the original formula in order to get better bounds, and possibly provide an accurate result for any quantifier tree.

References

Allen, J. F., Nov. 1983. Maintaining Knowledge About Temporal Intervals. Commun. ACM 26 (11), 832–843.

Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M. Y., Zbar, Y., 2002. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In: TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Vol. 2280 of Lecture Notes in Computer Science. Springer, Berlin, Germany, Grenoble, France, April 6–14, pp. 296–211.

Banieqbal, B., Barringer, H., 1987. Temporal Logic with Fixed Points. In: Banieqbal, B., Barringer, H., Pnueli, A. (Eds.), Temporal Logic in Specification. Vol. 398 of Lecture Notes in Computer Science. Springer, Altrincham, UK, April 8-10, pp. 62–74.

Bozzelli, L., Molinari, A., Montanari, A., Peron, A., Sala, P., 2016. Interval Temporal Logic Model Checking: The Border Between Good and Bad HS Fragments. In: Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings. Springer International Publishing, pp. 389–405.

Büchi, J. R., 1960. Weak Second-Order Arithmetic and Finite Automata. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6, 66–92.

Cerna, D., October 2015a. Space Complexity of LogicGuard Revisited. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

Cerna, D., May 2015b. Space Complexity of Operational Semantics for the LogicGuard Core Language. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz.

Cerna, D. M., Schreiner, W., Kutsia, T., 2016a. Better space bounds for future-looking stream monitors. Journal of Symbolic ComputationSubmitted.

Cerna, D. M., Schreiner, W., Kutsia, T., 2016b. Predicting Space Requirements for a Stream Monitor Specification Language. In: Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 28-30, 2016. Proceedings. Springer International Publishing, pp. 135–151.

Cerna, D. M., Schreiner, W., Kutsia, T., 2016c. Space Analysis of a Predicate Logic Fragment for the Specification of Stream Monitors. In: Davenport, J. H., (ed.), F. G. (Eds.), Proceedings of The 7th International Symposium on Symbolic Computation in Software Science. Vol. 39 of EPiC Series in Computing. pp. 29–41.

Finkbeiner, B., Kuhtz, L., 2009. Monitor Circuits for LTL with Bounded and Unbounded Future. In: Runtime Verification, 9th International Workshop, RV 2009. Vol. 5779 of Lecture Notes in Computer Science. Springer, Berlin, Grenoble, France, June 26–28, pp. 60–75.

Frick, M., Grohe, M., 2004. The Complexity of First-Order and Monadic Second-Order Logic Revisited. Annals of Pure and Applied Logic 130 (1–3), 3–31.

Gottlob, G., Mar. 1995. NP Trees and Carnap's Modal Logic. J. ACM 42 (2), 421–457.

Halpern, J. Y., Shoham, Y., Oct. 1991. A Propositional Modal Logic of Time Intervals. Journal of the ACM (JACM) 38 (4), 935–962.

IEEE, 2007. IEEE Std 1850-2007: Standard for Property Specification Language (PSL).

Kupferman, O., Lustig, Y., Vardi, M. Y., 2006. On Locally Checkable Properties. In: Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006. Vol. 5779 of Lecture Notes in Artificial Intelligence. Springer, Berlin, Germany, Phnom Penh, Cambodia, November 13–17, pp. 302–316.

Kutsia, T., Schreiner, W., 2014. Verifying the Soundness of Resource Analysis for LogicGuard Monitors (Revised Version). Technical Report 14-08, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

LogicGuard II, November 2015. LogicGuard II. `http://www.risc.jku.at/projects/LogicGuard2/`.

Maler, O., Nickovic, D., Pnueli, A., 2005. Real Time Temporal Logic: Past, Present, Future. In: Pettersson, P., Yi, W. (Eds.), Formal Modeling and Analysis of Timed Systems, Third International Conference (FORMATS). Vol. 3829 of Lecture Notes in Computer Science. Springer, Berlin, Germany, Uppsala, Sweden, September 26–28, pp. 2–16.

McNaughton, R., Papert, S., 1971. Counter-Free Automata. Vol. 65 of Research Monograph. MIT Press, Cambridge, MA, USA.

Molinari, A., Montanari, A., Peron, A., 2015. A Model Checking Procedure for Interval Temporal Logics based on Track Representatives. In: Kreutzer, S. (Ed.), 24th EACSL Annual Conference on Computer Science Logic (CSL 2015). Vol. 41 of Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 193–210.

Rosu, G., Bensalem, S., 2006. Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis. In: Ball, T., Jones, R. B. (Eds.), Computer Aided Verification, 18th International Conference, (CAV). Vol. 4144 of Lecture Notes in Computer Science. Springer, Berlin, Germany, Seattle, WA, USA, August 17–20, pp. 263–277.

Schnoebelen, P., 2003a. Oracle Circuits for Branching-Time Model Checking. In: Baeten, J. C. M., Lenstra, J. K., Parrow, J., Woeginger, G. J. (Eds.), Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30 – July 4, 2003 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 790–801.

Schnoebelen, Ph., 2003b. The Complexity of Temporal Logic Model Checking. In: Balbiani, Ph., Suzuki, N.-Y., Wolter, F., Zakharyaschev, M. (Eds.), Selected Papers from the 4th Workshop on Advances in Modal Logics (AiML'02). King's College Publication, Toulouse, France, pp. 393–436, invited paper.

Schreiner, W., Kutsia, T., Cerna, D., Krieger, M., Ahmad, B., Otto, H., Rummerstorfer, M., Gössl, T., November 2015. The LogicGuard Stream Monitor Specification Language (Version 1.01). Tutorial and reference manual, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

Vardi, M. Y., Wolper, P., 1986. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In: Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18. IEEE Computer Society, pp. 332–344.