Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Expressive Logical Combinators for Free

Pierre Genevès
CNRS, LIG

Alan Schmitt*
Inria, Rennes

Abstract

A popular technique for the analysis of web query languages relies on the translation of queries into logical formulas. These formulas are then solved for satisfiability using an off-the-shelf satisfiability solver. A critical aspect in this approach is the size of the obtained logical formula, since it constitutes a factor that affects the combined complexity of the global approach. We present logical combinators whose benefit is to provide an exponential gain in succinctness in terms of the size of the logical representation. This opens the way for solving a wide range of problems such as satisfiability and containment for expressive query languages in exponential-time, even though their direct formulation into the underlying logic results in an exponential blowup of the formula size, yielding an incorrectly presumed two-exponential time complexity. We illustrate this from a practical point of view on a few examples such as numerical occurrence constraints and tree frontier properties which are concrete problems found with semi-structured data.

1 Introduction

Modal logics have recently been increasingly used in the analysis of domain specific languages such as XML Schemas [Dal-Zilio *et al.*, 2004; Bárcenas *et al.*, 2011], XPath queries [Genevès *et al.*, 2007], SPARQL queries [Chekol *et al.*, 2012], and CSS selectors [Genevès *et al.*, 2012]. They also serve as a formal framework for the analysis of programming languages such as CDuce [Benzaken *et al.*, 2003; Gesbert *et al.*, 2011], XQuery [Boag *et al.*, 2005; Castagna and Xu, 2011], languages for XML access control [Murata *et al.*, 2006; Kolovski *et al.*, 2007] and, more generally, for NoSQL programming languages [Gesbert *et al.*, 2011; Benzaken *et al.*, 2013]. The common goal is to characterize these languages in terms of expressivity and complexity, and to build adapted and effective type-checkers, static analyzers, and compilers.

Such an approach requires the construction of efficient compilers that translate first-class constructs (such as queries) into logical formulas. Those formulas are then solved for satisfiability using an off-the-shelf satisfiability solver such as the ones found in [Tanabe *et al.*, 2005; Pan *et al.*, 2006; Genevès *et al.*, 2007; Genevès *et al.*, 2015a; Tanabe *et al.*, 2008]. A critical aspect is then the size of the obtained logical formula, since it is a factor that affects the combined complexity[1] of the global approach.

We show that a whole class of logical combinators (or "macros") can be used as an intermediate language between the query language and the logical language. Those logical combinators provide an exponential gain in succinctness over the corresponding explicit logical representation, yet preserving the typical exponential time complexity [Tanabe *et al.*, 2005; Pan *et al.*, 2006; Genevès *et al.*, 2007; Genevès *et al.*, 2015a; Tanabe *et al.*, 2008] of the logical decision procedure.

This opens the way for solving a wide range of problems such as query satisfiability and query containment in exponential-time, even though their direct formulation into the underlying logic results in an exponential blowup of the formula size, yielding an incorrectly presumed two-exponential time complexity.

Specifically, two essential steps are involved in the reduction of a problem to logical satisfiability: (1) the translation of the initial problem into a logical formula, and (2) the actual satisfiability check of the formula. Traditionally, the complexity of the satisfiability test is stated in terms of the size of the formula, thus every duplication of subformulas during the first step may affect the combined complexity and severely impact the practical applicability of the entire approach. Interestingly, we observe that a common form of μ-calculus sub-formula duplication has a very limited impact on combined complexity in existing implementations, such as [Tanabe *et al.*, 2005; Pan *et al.*, 2006; Genevès *et al.*, 2015a; Tanabe *et al.*, 2008]. The reason lies in the fact that satisfiability-testing algorithms can operate directly on a Hintikka-set-like representation of formulas,

*Detailed affiliation of authors: P. Genevès (CNRS, Laboratoire d'Informatique de Grenoble, Inria, Univ. Grenoble Alpes) and A. Schmitt (Inria, Research Center Rennes - Bretagne Atlantique).

[1] In the context of problem-solving by reduction to logical satisfiability, combined complexity considers the complexity of the translation of the problem into logic (taking into account any potential blow-up in size induced by the change in representation), composed with the complexity of testing satisfiability of the formula.

named lean, composed of atomic propositions and modal sub-formulas. In other terms, the lean can be seen as a succinct but sufficient representation from which the logical satisfiability of the formula can be decided. In this setting, we prove that the time complexity of decision procedures actually depends on the number of *distinct* atomic propositions and modal sub-formulas present in the lean. This makes explicit a notion of truth-status sharing for identical sub-formulas not exhibited in the analysis of the time complexity of such algorithms.

We develop this idea in the context of the μ-calculus, whose expressive power subsumes the ones of many modal logics. More specifically we develop this idea using the alternation-free μ-calculus with converse modalities whose models are finite trees, following [Genevès *et al.*, 2007]. Trees are encoded in binary, without loss of generality, through the "first-child" and "next-sibling" modalities, respectively noted $\langle 1 \rangle$ and $\langle 2 \rangle$ in the manner of [Genevès *et al.*, 2007].

In this setting, an elegant way of building a μ-calculus formula is to apply a combinator to another formula. For instance, $\texttt{split}(X) = \langle 1 \rangle X \wedge \langle 2 \rangle X$ is a combinator that generates a formula such that the input formula must hold in both successors of the current node. In the previous example, although X is duplicated, the increase of the size of the lean generated from $\texttt{split}(\varphi)$ when compared to the one generated from φ is only a small constant, independent of φ.

The paper is organized as follows. We introduce the logical formulas and combinators in §2, state and prove our main result in §3. Finally, we report on practical applications in §4 with experimental results that illustrate that the lean size is indeed a better indicator of the problem's complexity than formula size.

2 Basic Logical Formulas and Combinators

We recall the syntax of the logic of [Genevès *et al.*, 2007], used to prove properties about finite binary trees. We consider a set AP of atomic propositions, representing the tree node names, which includes a special reserved name $\not\sigma$; a set Var of variables, used in fixpoints; and a set $\text{Prog} = \{1, 2, \overline{1}, \overline{2}\}$ of programs, to describe navigation in a tree. Program 1 navigates to the first child (left successor), program 2 navigates to the next sibling (right successor), program $\overline{1}$ to the parent (predecessor to the right, if it exists), and program $\overline{2}$ to the previous sibling (predecessor to the left, if it exists). We let $\overline{\overline{a}} = a$ for any $a \in \text{Prog}$. A logical formula is defined using the following syntax.

- $\top$, $\bot$, σ, or $\neg\sigma$ for all $\sigma \in \text{AP} \setminus \{\not\sigma\}$;
- x for all $x \in \text{Var}$;
- $\varphi_1 \vee \varphi_2$ or $\varphi_1 \wedge \varphi_2$ where φ_1 and φ_2 are logical formulas;
- $\langle a \rangle \varphi$ or $\neg \langle a \rangle \top$ where $a \in \text{Prog}$ and φ is a logical formula;
- $\mu x.\varphi$ where $x \in \text{Var}$ and φ is a logical formula.

We now give an intuition of the interpretation of formulas in the setting of finite trees: the interpretation of a formula is a set of *focused trees*, which are finite trees with a selected node. A formula is *satisfiable* if there exists a tree such that a node of this tree is *selected* by the formula (i.e., the set of focused trees is not empty). The truth formula $\top$ selects all focused trees, i.e., every node of every tree, whereas $\bot$ selects none. The σ formula selects every focused trees whose selected node's name is σ, whereas $\neg\sigma$ selects the nodes with other names (the node name $\not\sigma$ is used to represent names of nodes not occurring in the formula). Formula conjunction and disjunction correspond to set intersection and union, respectively. A formula $\langle a \rangle \varphi$ selects a node if the node reached following a is selected by formula φ. Formula $\neg \langle a \rangle \top$ selects a node if there is no node reachable through a. Finally, a fixpoint $\mu x.\varphi$ is interpreted as the smallest fixpoint (the intersection of every pre-fixpoint).

A formula is *closed* if every occurrence of a variable x is bound by an enclosing μx. In the following, we only consider formulas that are closed and whose recursion variables are *guarded* (there is at least one navigation step between a recursion μx and every variable x). Note that since we do not have general negation in formulas (see below), there is no requirement for formulas to be positive (i.e., disallow formulas of the form $\mu x. \ldots \neg x$): such formulas simply cannot be expressed. Finally, we write $\varphi \prec \psi$ if φ is a sub-formula of ψ, and $\varphi \not\prec \psi$ if it is not.

A *combinator* F is a formula with zero or more occurrences of a placeholder, written X, possibly negated ($\neg X$). We write $F\{\varphi/X\}$ for the combinator F where every instance of X has been replaced by the closed formula φ.[2] We often write $F(X)$ to make clear the name of the placeholder, and $F(\varphi)$ for $F\{\varphi/X\}$.

We consider formulas in negation normal form. The negation of a formula or combinator, written $\overline{F}$, is obtained using regular De Morgan's laws extended with the cases of fixpoints: $\overline{\mu x.F} = \mu x.\overline{F}$, $\overline{x} = x$, $\overline{X} = \neg X$, and $\overline{\neg X} = X$. Moreover, the negation of a modality is a disjunction:

$$\overline{\langle a \rangle F} = \langle a \rangle \overline{F} \vee \neg \langle a \rangle \top \quad F \neq \top$$

the modality is false either because there is no node in that direction, or because the node in that direction does not satisfy the sub-formula. Note that, in order to ensure that $\overline{\overline{F}} = F$ for every formula, we have a special case when a disjunction is actually the negation of a modality:

$$\overline{\langle a \rangle F \vee \neg \langle a \rangle \top} = \langle a \rangle \overline{F}$$

Following [Genevès *et al.*, 2007], the greatest and smallest fixpoint coincide (provided a simple restriction on formulas, namely for cycle-free formulas using sets of finite trees as models, see [Genevès *et al.*, 2007] for details). Every combinator presented here fulfill this restriction. Nevertheless, this work could also be done in a setting where the smallest and greatest fixpoints differ, and in this case one defines $\overline{\mu x.F}$ as $\nu x.\overline{F}$ and $\nu x.\overline{F}$ as $\mu x.\overline{F}$.

3 Deciding Combined Formulas

The Lean

Following [Pan *et al.*, 2006; Genevès *et al.*, 2007; Tanabe *et al.*, 2008], we define the lean of F as follows, with $\overline{\mathcal{L}}_\Gamma(F)$

[2] In case of a negated placeholder, we replace $\neg X$ with $\overline{\varphi}$, the negation normal form of φ (obtained using De Morgan's laws).

defined in Figure 1 (the environment Γ is the set of already unfolded fixpoints). The main difference with the usual approaches is that we close the lean under negation.

$$\text{Lean}(F) = \left\{ \langle a \rangle \top \mid a \in \{1, 2, \overline{1}, \overline{2}\} \right\} \cup \{\diamond\!\!\!\!-\} \cup \overline{\mathcal{L}}_\emptyset(F)$$

$$\begin{gathered}
\overline{\mathcal{L}}_\Gamma(\top) = \overline{\mathcal{L}}_\Gamma(\bot) = \overline{\mathcal{L}}_\Gamma(x) = \overline{\mathcal{L}}_\Gamma(X) = \overline{\mathcal{L}}_\Gamma(\neg X) = \emptyset \\
\overline{\mathcal{L}}_\Gamma(\sigma) = \overline{\mathcal{L}}_\Gamma(\neg\sigma) = \{\sigma\} \\
\overline{\mathcal{L}}_\Gamma(F \vee G) = \overline{\mathcal{L}}_\Gamma(F \wedge G) = \overline{\mathcal{L}}_\Gamma(F) \cup \overline{\mathcal{L}}_\Gamma(G) \\
\overline{\mathcal{L}}_\Gamma(\langle a \rangle F) = \{\langle a \rangle F, \langle a \rangle \overline{F}, \langle a \rangle \top\} \cup \overline{\mathcal{L}}_\Gamma(F) \\
\overline{\mathcal{L}}_\Gamma(\neg \langle a \rangle \top) = \{\langle a \rangle \top\} \\
\overline{\mathcal{L}}_\Gamma(\mu x.F) = \overline{\mathcal{L}}_\Gamma(F) \quad \text{if } x \not\prec F \\
\overline{\mathcal{L}}_\Gamma(\mu x.F) = \emptyset \quad \text{if } \mu x.F \in \Gamma \text{ or } \mu x.\overline{F} \in \Gamma \\
\overline{\mathcal{L}}_\Gamma(\mu x.F) = \overline{\mathcal{L}}_{\Gamma \cup \{\mu x.F\}}(F\{^{\mu x.F}/_x\}) \quad \text{otherwise}
\end{gathered}$$

Figure 1: Negation Closed Lean

Intuitively, the lean of a formula φ is the set of every atomic proposition occurring in φ, and every subformula that starts with a modality present in φ or in the expansion of the fixpoints of φ. In particular, the lean does not directly include disjunctive or conjunctive subformulas. Kozen has shown in [Kozen, 1982] that expanding every fixpoint once is sufficient to generate every subformula that may need to be considered for satisfiability. This single expansion is tracked using the Γ argument in the definition above. Moreover, Kozen has also shown the lean is linear in the size of the formula. We next make this bound more precise.

The Factorization Power of the Lean

We can now state the main theorem of this paper: the lean size is not impacted by the duplication of sub-formulas. We write $|S|$ for the size of a set S.

Theorem 1. *Let F be a combinator and φ a closed formula. We have the following.*

$$|\mathit{Lean}(F\{^{\varphi}/_X\})| \leq |\mathit{Lean}(F)| + |\mathit{Lean}(\varphi)|$$

The theorem is a direct consequence of Lemma 9 which is proved below in §3.

We now give some intuition about this result, through a simple example. Recall the $\texttt{split}(X)$ combinator defined as $\langle 1 \rangle X \wedge \langle 2 \rangle X$. Since the elements of the lean are either atomic propositions (node names) and modalities, the lean of $\texttt{split}(\varphi)$ includes the lean of φ and four new elements: $\langle 1 \rangle \varphi$, $\langle 1 \rangle \overline{\varphi}$, $\langle 2 \rangle \varphi$, and $\langle 2 \rangle \overline{\varphi}$. If we now consider $\texttt{split}(\texttt{split}(\varphi))$, we once again add only four formulas to the lean: $\langle 1 \rangle \texttt{split}(\varphi)$, $\langle 1 \rangle \overline{\texttt{split}(\varphi)}$, $\langle 2 \rangle \texttt{split}(\varphi)$, and $\langle 2 \rangle \overline{\texttt{split}(\varphi)}$. This linear growth, even though the formula's size increases exponentially, is due to the fact that modalities are considered atomically and are not split up in their components (e.g., $\langle 1 \rangle (\varphi \wedge \psi)$ is not split up into $\langle 1 \rangle \varphi$ and $\langle 1 \rangle \psi$).

Satisfiability-Testing Algorithms based on the Lean

A typical approach to decide the satisfiability of a formula is to first build the lean, as described above, then to use a tableau-based algorithm implemented with BDDs [Tanabe *et al.*, 2005; Pan *et al.*, 2006; Genevès *et al.*, 2007; Tanabe *et al.*, 2008]. The time complexity of this approach is shown to be exponential in the size of the formula. More precisely, it is exponential in the size of the lean, which is in turn linear in the size of the formula.

The essence of this paper is to show (both in theory and in practice) that the lean may grow much more slowly than the formula when sub-formulas are duplicated. This opens the way for solving a wide range of problems in exponential-time even though their direct translation into the modal logic is exponential, as illustrated on concrete examples in §4.

Proof of Theorem 1

We define the *number of recursive expansions* of F or φ, written $\mathcal{E}_\emptyset(F)$, in Figure 2. We use this number to define inductive properties that depend on fixpoints being expanded.

$$\begin{gathered}
\mathcal{E}_\Gamma(\top) = \mathcal{E}_\Gamma(\bot) = \mathcal{E}_\Gamma(x) = \mathcal{E}_\Gamma(X) = \mathcal{E}_\Gamma(\neg X) = 0 \\
\mathcal{E}_\Gamma(\sigma) = \mathcal{E}_\Gamma(\neg\sigma) = \mathcal{E}_\Gamma(\neg \langle a \rangle \top) = 0 \\
\mathcal{E}_\Gamma(F \vee G) = \mathcal{E}_\Gamma(F \wedge G) = \mathcal{E}_\Gamma(F) + \mathcal{E}_\Gamma(G) \\
\mathcal{E}_\Gamma(\langle a \rangle F) = \mathcal{E}_\Gamma(F) \\
\mathcal{E}_\Gamma(\mu x.F) \stackrel{\text{def}}{=} \mathcal{E}_\Gamma(F) \quad \text{if } x \not\prec F \\
\mathcal{E}_\Gamma(\mu x.F) \stackrel{\text{def}}{=} 0 \quad \text{if } \mu x.F \in \Gamma \text{ or } \mu x.\overline{F} \in \Gamma \\
\mathcal{E}_\Gamma(\mu x.F) \stackrel{\text{def}}{=} \mathcal{E}_{\Gamma \cup \{\mu x.F\}}(F\{^{\mu x.F}/_x\}) + 1 \quad \text{otherwise}
\end{gathered}$$

Figure 2: Number of Recursive Expansions

We write $(\Gamma)^{\varphi}_X$ as the set of formulas G' where either $G'\{^{\varphi}/_X\}$ or $\overline{G'}\{^{\varphi}/_X\}$ is in Γ. We use this set to identify the formulas that were expanded.

Definition 2. *Given φ, and Γ, we define $(\Gamma)^{\varphi}_X$ as follows.*

$$\{G' \mid X \prec G' \wedge (G'\{^{\varphi}/_X\} \in \Gamma \vee \overline{G'}\{^{\varphi}/_X\} \in \Gamma)\}$$

Lemma 3. *We have $\overline{F\{^{G}/_x\}} = \overline{F}\{^{\overline{G}}/_x\}$.*

Proof. By induction on F, relying on the fact that $\overline{\overline{x}} = x$. □

Lemma 4. *We have $\overline{F\{^{G}/_X\}} = \overline{F}\{^{G}/_X\}$.*

Proof. By induction on F, relying on the fact that $\overline{\overline{X}} = \neg X$. □

Lemma 5. *If $\Gamma \subseteq \Gamma'$, then $\overline{\mathcal{L}}_{\Gamma'}(F) \subseteq \overline{\mathcal{L}}_\Gamma(F)$.*

Proof. By induction on the lexical order of $\mathcal{E}_\Gamma(F)$ and the size of F. Base cases are immediate. For conjunction and disjunction, we may apply the induction hypothesis because $\mathcal{E}_\Gamma(F_1)$ and $\mathcal{E}_\Gamma(F_2)$ do not increase and the formula size decreases. This is also the case for the modality case.

For the recursion case where $x \prec F$, we distinguish three cases (we do not mention the negation $\mu x.\overline{F}$ in these cases):

- if $\mu x.F \in \Gamma$, then necessarily $\mu x.F \in \Gamma'$ and we immediately conclude;
- if $\mu x.F \in \Gamma'$, then we conclude by $\emptyset \subseteq \mathcal{S}$ for any set $\mathcal{S}$;
- otherwise, we have $\mu x.F$ in neither set, and we apply the induction hypothesis, as $\mathcal{E}_{\Gamma\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\})$ is strictly smaller than $\mathcal{E}_\Gamma(\mu x.F)$.

□

Lemma 6. *We have* $\overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(G) = \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.\overline{F}\}}(G)$.

Proof. By an immediate induction on the lexical order of $\mathcal{E}_\Gamma(G)$ and the size of G. □

Lemma 7. *For any F, we have* $\overline{\mathcal{L}}_\Gamma(F) = \overline{\mathcal{L}}_\Gamma(\overline{F})$.

Proof. By induction on the lexical order of $\mathcal{E}_\Gamma(F)$ and the size of F.

The base cases $\top$, $\bot$, x, X, $\neg X$, σ, $\neg\sigma$, $\neg\langle a\rangle\top$, and $\mu x.F$ there $x \neq F$ are immediate.

For $F \wedge G$, we compute as follows.

$$\begin{aligned}\overline{\mathcal{L}}_\Gamma(F\wedge G) &= \overline{\mathcal{L}}_\Gamma(F)\cup\overline{\mathcal{L}}_\Gamma(G)\\ &= \overline{\mathcal{L}}_\Gamma(\overline{F})\cup\overline{\mathcal{L}}_\Gamma(\overline{G}) && \text{by induction}\\ &= \overline{\mathcal{L}}_\Gamma(\overline{F}\vee\overline{G})\\ &= \overline{\mathcal{L}}_\Gamma(\overline{F\wedge G})\end{aligned}$$

The disjunction case is similar.

For the recursion case, if $\mu x.F$ or $\mu x.\overline{F}$ is in Γ, then $\mu x.\overline{F}$ or $\mu x.\overline{\overline{F}} = \mu x.F$ is also in Γ and the result follows.

Finally, if neither is in Γ, we compute as follows.

$$\begin{aligned}\overline{\mathcal{L}}_\Gamma(\mu x.F) &= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\})\\ &= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(\overline{F\{^{\mu x.F}/_x\}}) && \text{by induction}\\ &= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(\overline{F}\{^{\mu x.\overline{F}}/_x\}) && \text{by Lemma 3}\\ &= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.\overline{F}\}}(\overline{F}\{^{\mu x.\overline{F}}/_x\}) && \text{by Lemma 6}\\ &= \overline{\mathcal{L}}_\Gamma(\mu x.\overline{F})\end{aligned}$$

□

Lemma 8. *For all Γ and F, if $X \neq F$, then* $\overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F) = \overline{\mathcal{L}}_\emptyset(F)$.

Proof. We prove the more general result: for all Γ', $\overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\Gamma'}(F) = \overline{\mathcal{L}}_{\Gamma'}(F)$ by induction on the lexical order of $\mathcal{E}_{\Gamma'}(F)$ and the size of F.

The result is immediate for the base cases, and by induction for the conjunction, disjunction, and modality cases. For the recursion case, if $\mu x.F$ (or its negation) is in $(\Gamma)^\varphi_X \cup \Gamma'$, then it must be in Γ' as $X \neq F$ and members of $(\Gamma)^\varphi_X$ contain X by definition. Thus both sides are equal to $\emptyset$. If neither $\mu x.F$ nor its negation are in $(\Gamma)^\varphi_X\cup\Gamma'$, we have $\overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\Gamma'}(\mu x.F) = \overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\Gamma'\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\})$

We next apply the induction hypothesis with $\Gamma' \cup \{\mu x.F\}$ and $F\{^{\mu x.F}/_x\}$, thus we have $\overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\Gamma'\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\}) = \overline{\mathcal{L}}_{\Gamma'\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\}) = \overline{\mathcal{L}}_{\Gamma'}(\mu x.F)$.

We conclude by taking $\Gamma' = \emptyset$. □

Lemma 9. *Let F be a formula mentioning X, and φ a closed formula. We have* $\overline{\mathcal{L}}_\emptyset(F\{^\varphi/_X\}) \subseteq \overline{\mathcal{L}}_\emptyset(F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)$.

Proof. We prove the following more general property for any Γ by induction on the lexical order of $\mathcal{E}_{(\Gamma)^\varphi_X}(F)$ and the size of F.

$$\overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\}) \subseteq \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)$$

We first deal with every case where $X \neq F$. In this case, X also does not occur in $\overline{\mathcal{L}}_\Gamma(F)$.

$$\begin{aligned}\overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\}) &= \overline{\mathcal{L}}_\Gamma(F)\\ &\subseteq \overline{\mathcal{L}}_\emptyset(F) && \text{by Lemma 5}\\ &= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F) && \text{by Lemma 8}\\ &= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\}\\ &\subseteq \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)\end{aligned}$$

Case X. We compute as follows, using Lemma 5 for the last inclusion.

$$\overline{\mathcal{L}}_\Gamma(X\{^\varphi/_X\}) = \overline{\mathcal{L}}_\Gamma(\varphi) \subseteq \overline{\mathcal{L}}_\emptyset(\varphi)$$

Case $\neg X$. We compute as follows, using Lemma 5 for the set inclusion, and Lemma 7 to conclude.

$$\overline{\mathcal{L}}_\Gamma((\neg X)\{^\varphi/_X\}) = \overline{\mathcal{L}}_\Gamma(\overline{\varphi}) \subseteq \overline{\mathcal{L}}_\emptyset(\overline{\varphi}) = \overline{\mathcal{L}}_\emptyset(\varphi)$$

Case $F \wedge G$. We compute as follows.

$$\begin{aligned}&\overline{\mathcal{L}}_\Gamma((F\wedge G)\{^\varphi/_X\})\\ &= \overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\})\cup\overline{\mathcal{L}}_\Gamma(G\{^\varphi/_X\})\\ &\quad \text{and by induction:}\\ &\subseteq \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(G)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)\\ &= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F\wedge G)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)\end{aligned}$$

Case $F \vee G$. Identical to the previous case.

Case $\langle a\rangle F$. We compute as follows, using Lemma 4 and the induction hypothesis.

$$\begin{aligned}&\overline{\mathcal{L}}_\Gamma((\langle a\rangle F))\{^\varphi/_X\}\\ &= \overline{\mathcal{L}}_\Gamma(\langle a\rangle(F\{^\varphi/_X\}))\\ &= \{\langle a\rangle F\{^\varphi/_X\}; \langle a\rangle\overline{F\{^\varphi/_X\}}; \langle a\rangle\top\}\cup\overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\})\\ &= \{\langle a\rangle F; \langle a\rangle\overline{F}; \langle a\rangle\top\}\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\})\\ &\subseteq \{\langle a\rangle F; \langle a\rangle\overline{F}; \langle a\rangle\top\}\{^\varphi/_X\}\cup\overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)\\ &= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(\langle a\rangle F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi)\end{aligned}$$

Case $\mu x.F$ with $x \neq F$. We compute as follows.

$$\begin{aligned}&\overline{\mathcal{L}}_\Gamma((\mu x.F))\{^\varphi/_X\}\\ &= \overline{\mathcal{L}}_\Gamma(\mu x.F\{^\varphi/_X\}) && \varphi \text{ closed}\\ &= \overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\}) && x \neq F\{^\varphi/_X\}\\ &\subseteq \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi) && \text{by induction}\\ &= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(\mu x.F)\{^\varphi/_X\}\cup\overline{\mathcal{L}}_\emptyset(\varphi) && x \neq F\end{aligned}$$

Case $\mu x.F$ with $x \prec F$.

If we have $\mu x.F\{\varphi/X\} \in \Gamma$ or $\mu x.\overline{F\{\varphi/X\}} \in \Gamma$ then $\overline{\mathcal{L}}_\Gamma(\mu x.F\{\varphi/X\}) = \emptyset$ and the result is immediate.

Otherwise we compute as follows.

$$\begin{aligned}
&\overline{\mathcal{L}}_\Gamma((\mu x.F)\{\varphi/X\}) \\
&= \overline{\mathcal{L}}_\Gamma(\mu x.F\{\varphi/X\}) && \varphi \text{ closed} \\
&= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\{\varphi/X\}\}}(F\{\varphi/X\}\{^{\mu x.F\{\varphi/X\}}/_x\}) \\
&= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\{\varphi/X\}\}}(F\{^{\mu x.F}/_x\}\{\varphi/X\}) && \varphi \text{ closed}
\end{aligned}$$

To apply the induction hypothesis, we show that

$$\mathcal{E}_{(\Gamma)^\varphi_X}(\mu x.F) = \mathcal{E}_{(\Gamma\cup\{\mu x.F\{\varphi/X\}\})^\varphi_X}(F\{^{\mu x.F}/_x\}) + 1.$$

First, we have $\mu x.F \notin (\Gamma)^\varphi_X$ and $\mu x.\overline{F} \notin (\Gamma)^\varphi_X$, since otherwise, we would have $\mu x.F\{\varphi/X\} \in \Gamma$ or $\mu x.\overline{F}\{\varphi/X\} = \mu x.\overline{F\{\varphi/X\}} \in \Gamma$, which we assumed to be false.
Thus $\mathcal{E}_{(\Gamma)^\varphi_X}(\mu x.F) = \mathcal{E}_{(\Gamma)^\varphi_X\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\}) + 1$. Next, we have $(\Gamma \cup \{\mu x.\overline{F}\{\varphi/X\}\})^\varphi_X = (\Gamma)^\varphi_X \cup \{\mu x.F\}$ by definition. Thus we have $\mathcal{E}_{(\Gamma)^\varphi_X}(\mu x.F) = \mathcal{E}_{(\Gamma\cup\{\mu x.F\{\varphi/X\}\})^\varphi_X}(F\{^{\mu x.F}/_x\}) + 1$.

We may thus apply the induction hypothesis and continue to compute.

$$\begin{aligned}
&\subseteq \overline{\mathcal{L}}_{(\Gamma\cup\{\mu x.F\{\varphi/X\}\})^\varphi_X}(F\{^{\mu x.F}/_x\})\{\varphi/X\} \cup \overline{\mathcal{L}}_\emptyset(\varphi) \\
&= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\})\{\varphi/X\} \cup \overline{\mathcal{L}}_\emptyset(\varphi)
\end{aligned}$$

As neither $\mu x.F$ nor $\mu x.\overline{F}$ are in $(\Gamma)^\varphi_X$, we have the following equality: $\overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(\mu x.F) = \overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\})$. We may thus conclude the computation as follows.

$$= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(\mu x.F)\{\varphi/X\} \cup \overline{\mathcal{L}}_\emptyset(\varphi)$$

We complete the proof by taking Γ to be $\emptyset$ and remarking that $(\emptyset)^\varphi_X = \emptyset$. □

4 Applications

We provide an implementation of the proposed system [Genevès *et al.*, 2015b]. Each example given in this section comes in a boxed version that can directly be used with the implementation [Genevès *et al.*, 2015b]. Tests are thus reproducible.

We present several examples where advanced properties on the underlying data structure (here a tree) can be formulated using combinators, and for which our result applies. In particular, after an introductory example, we consider numerical constraints on the global number of occurrences, and properties on the sequence of leaves in a tree. This means, for instance, that if one extends a query language (such as XPath) with such kinds of features, then our result applies: problems such as query satisfiability and query containment would not be harder to solve in terms of computational complexity than they already are for the language without the extensions.

A Very Simple Example: Split

The combinator $\texttt{split}(X)$ introduced in Section 3 may generate arbitrary large formulas: for instance, let $\varphi = a \wedge \langle 1\rangle\, b \wedge \langle 2\rangle\, \mu y.c \vee \langle 2\rangle\, y$, the formula $\psi = \texttt{split}(\texttt{split}(\texttt{split}(\varphi)))$ uses 8 occurrences of φ. To give this formula to the implementation of [Genevès *et al.*, 2015b], we write it as follows.

```
phi() = a & <1>b & <2>let $y = c | <2>$y in $y;
split(#x) = <1>#x & <2>#x;
split(split(split(phi())))
```

We then observe that ψ contains 24 atomic propositions, 38 modalities, 15 conjunctions, and 8 disjunctions (including duplicates). The size of the lean is only 19 (14 modalities and 5 atomic propositions[3] Each new $\texttt{split}(\_)$ around the formula then only adds two elements to the lean.

The satisfiability check of the above formula is performed in 131ms (milliseconds) with the implementation [Genevès *et al.*, 2015b], including 5ms for computing the lean and 104ms for computing the tableau. A sample satisfying tree of 33 nodes is also constructed in 39ms.

Document-Order Relation and Global Counting

Extending modal logics with operators for counting occurrences of specific nodes in trees is a notably difficult problem [Lugiez, 2005; Dal-Zilio *et al.*, 2004]. In this example we review how our result applies for a simple form of counting (with respect to constants) in trees.

A very simple example of a combinator is the descendant relation that checks that a node satisfying some formula X is accessible in the subtree by any sequence of forward modalities. It is encoded as follows:

$$\texttt{descendant}(X) = \langle 1\rangle\, (\mu z.X \vee \langle 1\rangle\, z \vee \langle 2\rangle\, z)$$

A whole range of combinators to navigate in a tree can be defined in a similar manner. In particular we can encode:

$$\texttt{following}(X) = \texttt{ancestor\_or\_self}(\psi) \quad \text{where}$$

$$\psi = \texttt{following\_sibling}(\texttt{descendant\_or\_self}(X))$$

These combinators are easily defined in our logic; as they are very commonly used, they are predefined in [Genevès *et al.*, 2015b]. They can be used as such to encode the so-called *document-order* relation $\ll$. This relation corresponds to the ordering of nodes given by a depth-first tree traversal: $x \ll y$ iff node y is visited after node x in a depth-first tree traversal. We define the combinator $\texttt{next}(X) = \texttt{descendant}(X) \vee \texttt{following}(X)$ with which we can mimic the document-order relation (we write $X \wedge \texttt{next}(Y)$ for $x \ll y$). Notice that this combinator duplicates formulas, since the placeholder X appears twice in its definition.

The document-order relation can be used to express global counting properties in trees. For instance, if we want to encode the so-called concept of a *nominal* – or more generally the fact that some formula ψ is satisfied by one and only one node in the tree – we can write:

```
psi() = a & <1>b & <2>let $y = c | <2>$y in $y;
next(#x) = descendant(#x) | following(#x);
previous(#x) = preceding(#x) | ancestor(#x);
nominal(#x) = #x & ~previous(#x) & ~next(#x);
nominal(psi())
```

[3]The numbers we report in this paper correspond to the numbers reported by the implementation [Genevès *et al.*, 2015b].

If we now want to force the existence of at least 4 different tree nodes that satisfy ψ, we can write:

```
psi() & next(psi() & next(psi() & next(psi())))
```

The full expansion of the above formula is notably large (if we count duplicates, the formula contains 468 atomic propositions, 871 modalities, 156 conjunctions, and 404 disjunctions). However, the size of its lean is 43 (38 modalities and 5 atomic propositions).

The satisfiability check of the latter formula above is performed in 205ms with the implementation [Genevès *et al.*, 2015b], including 15ms for computing the lean and 124ms for the tableau. A sample satisfying tree is built in 78ms.

The Tree Frontier

In this example, borrowed from [Afanasiev *et al.*, 2005], one describes constraints on a tree frontier. A tree frontier is the set of leaves (nodes without an outgoing "1" edge) ordered from left to right. A frontier node y is the successor of a frontier node x iff $x \ll y$ and there is no leaf node in between x and y in the document order. A simple case analysis shows that node y is the successor of a frontier node x in one of three cases:

1. Either x is a leaf with an immediate next sibling which is also a leaf (y);
2. or x is a leaf with an immediate next sibling which is not a leaf, in which case, by navigating downward in its subtree we reach the leftmost leaf (y);
3. or x is a leaf with no next sibling, in which case, by going up to the parent node recursively until we reach a parent node which has a next sibling, then going to this next sibling, and then, from this node, navigating downward in its subtree we reach the leftmost leaf (y).

This yields the following definition of a combinator that captures all the aforementioned cases with the help of a few neatly chosen auxiliary predicates:

$$\texttt{nextFrontierNode}(Y) = \texttt{leaf} \wedge \texttt{upUntilRsibl}(\psi)$$

In this definition, the placeholder Y is to be replaced by a formula that holds at the successor node, and:

$$\begin{aligned}
\texttt{leaf} &= \neg \langle 1 \rangle \top \\
\psi &= \langle 2 \rangle \, \texttt{down_to_first_leaf}(Y) \\
\texttt{upUntilRsibl}(X) &= (\langle 2 \rangle \top \wedge X) \\
&\quad \vee \mu x. \, \langle \overline{1} \rangle \, ((\langle 2 \rangle \top \wedge X) \vee x) \\
&\quad \vee \langle \overline{2} \rangle \, x \\
\texttt{down_to_first_leaf}(Z) &= \mu x.(\texttt{leaf} \wedge Z) \vee \langle 1 \rangle \, x
\end{aligned}$$

Using these combinators, we can now express properties on the tree frontier. For instance, the formula shown on Figure 3 states that the leftmost leaf is labeled "a", and, by further navigation on the tree frontier, we encounter two other leaves labeled "a". If we count duplicates, the corresponding formula contains 7 atomic propositions, 26 modalities, 23 variables, 10 fixpoint binders, 7 negations, 7 conjunctions, and 16 disjunctions. However, the size of the corresponding lean is 22. The lean is only composed of 19 distinct modalities and 3 distinct atomic propositions. Each additional nested call to the combinator $\_ \wedge \texttt{nextFrontierNode}(Y)$ extends the lean by 6 modalities. However, the corresponding global formula goes from 26 modalities to 58 for the first addition, then it goes to 122 for the second addition. It reaches 32762 modalities for the 10^{th} addition, whereas the corresponding formula is solved for satisfiability in 11052ms (lean size is 82).

```
leaf() = ~<1>T;
down_to_first_leaf(#z)= let $x= (leaf() & <0>#z)
  |<1>$x in $x;

up_until_rsibl(#x) = (<2>T & #x)
   | let $w = <-1>((<2>T & #x) | $w)
   | <-2>$w in $w;

next_frontier_node(#y) = leaf()
   & up_until_rsibl(<2>down_to_first_leaf(#y));

down_to_first_leaf(a & next_frontier_node(a
   & next_frontier_node(a)))
```

Figure 3: Tree Frontier Example.

The satisfiability check of the formula shown in Figure 3 is performed in 136ms with the implementation [Genevès *et al.*, 2015b], including 3ms for computing the lean and 109ms for computing the tableau. A sample satisfying tree is built in 18ms.

5 Conclusion

We have presented the concept of logical combinators that avoid exponential increases in combined complexity due to sub-formula duplication. Our main result, of theoretical nature, has very practical consequences and applies for a large class of logical solvers such as the ones found in [Tanabe *et al.*, 2005; Pan *et al.*, 2006; Genevès *et al.*, 2007; Genevès *et al.*, 2015a; Tanabe *et al.*, 2008].

We have further illustrated this result in the context of one of these satisfiability solvers [Genevès *et al.*, 2007; Genevès *et al.*, 2015a], for which we have presented an in-depth analysis. This analysis focuses on the time complexity of lean-based algorithms to decide the satisfiability of a tree logic equipped with inverse programs, nominals, and counting introduced via combinators. The analysis highlights our result by showing that the lean automatically factorizes duplicated sub-formulas even for such advanced features, thus the complexity of the algorithm should not be stated in terms of the size of the initial formula but in terms of the size of the lean. A direct consequence of this observation is that the addition of nominals and a more general form of counting to the initial tree logic has no impact on decidability nor on its precise complexity bound. We have also reported on practical experiments using an implementation.

References

[Afanasiev *et al.*, 2005] L. Afanasiev, P. Blackburn, I. Dimitriou, B. Gaiffe, E. Goris, M. Marx, and M. de Rijke. PDL for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135, 2005.

[Bárcenas *et al.*, 2011] Everardo Bárcenas, Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Query reasoning on trees with types, interleaving, and counting. In *IJCAI 2011: Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 718–723, Jul 2011.

[Benzaken *et al.*, 2003] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP'03: ICFP 11: Proceedings of the ACM international conference on Functional programming*, pages 51–63, 2003.

[Benzaken *et al.*, 2013] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of nosql languages. In *POPL'13: Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 101–114, 2013.

[Boag *et al.*, 2005] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language, 2005. http://www.w3.org/TR/xquery/.

[Castagna and Xu, 2011] Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ICFP'11: Proceedings of the 16th ACM SIGPLAN international conference on Functional Programming*, pages 94–106, 2011.

[Chekol *et al.*, 2012] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. SPARQL query containment under RDFS entailment regime. In *IJCAR: Proceedings of the 6th International Joint Conference on Automated Reasoning*, pages 134–148, 2012.

[Dal-Zilio *et al.*, 2004] Silvano Dal-Zilio, Denis Lugiez, and Charles Meyssonnier. A logic you can count on. In *POPL'04: Proceedings of the ACM Symposium on Principles of Programming Languages*, 2004.

[Genevès *et al.*, 2007] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351, 2007.

[Genevès *et al.*, 2012] Pierre Genevès, Nabil Layaïda, and Vincent Quint. On the analysis of cascading style sheets. In *WWW'12: Proceedings of the 21st World Wide Web Conference*, pages 809–818, April 2012.

[Genevès *et al.*, 2015a] Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. Efficiently Deciding μ-Calculus with Converse over Finite Trees. *ACM Transactions on Computational Logic*, 16(2), 2015.

[Genevès *et al.*, 2015b] Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. The XML reasoning project: http://wam.inrialpes.fr/websolver/, April 2015.

[Gesbert *et al.*, 2011] Nils Gesbert, Pierre Genevès, and Nabil Layaïda. Parametric polymorphism and semantic subtyping: the logical connection. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 107–116, 2011.

[Kolovski *et al.*, 2007] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 677–686, 2007.

[Kozen, 1982] D. Kozen. Results on the propositional μ-Calculus. In *ICALP'82: Proceedings of the International Colloquium on Automata, Languages, and Programming*, 1982.

[Lugiez, 2005] Denis Lugiez. Multitree automata that count. *Theor. Comput. Sci.*, 333(1-2):225–263, 2005.

[Murata *et al.*, 2006] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, 2006.

[Pan *et al.*, 2006] Guoqiang Pan, Ulrike Sattler, and Moshe Y. Vardi. BDD-based decision procedures for the modal logic K. *Journal of Applied Non-classical Logics*, 16(1-2):169–208, 2006.

[Tanabe *et al.*, 2005] Yoshinori Tanabe, Koichi Takahashi, Mitsuharu Yamamoto, Akihiko Tozawa, and Masami Hagiya. A decision procedure for the alternation-free two-way modal μ-calculus. In *TABLEAUX*, volume 3702 of *Lecture Notes in Computer Science*, pages 277–291, 2005.

[Tanabe *et al.*, 2008] Yoshinori Tanabe, Koichi Takahashi, and Masami Hagiya. A decision procedure for alternation-free modal μ–calculi. In *Advances in Modal Logic*, pages 341–362, 2008.