HAL
open science

Logical Combinators for Rich Type Systems

Pierre Genevès, Nabil Layaïda, Alan Schmitt

To cite this version:

Pierre Genevès, Nabil Layaïda, Alan Schmitt. Logical Combinators for Rich Type Systems. [Research Report] RR-8010, INRIA. 2012, pp.18. hal-00714353v2

HAL Id: hal-00714353

https://inria.hal.science/hal-00714353v2

Submitted on 4 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Logical Combinators for Rich Type Systems

Pierre Genevès, Nabil Layaïda, Alan Schmitt

RESEARCH REPORT
N° 8010
July 2012
Project-Teams Wam and Celtique

ISSN 0249-6399 ISRN INRIA/RR--8010--FR+ENG

Logical Combinators for Rich Type Systems

Pierre Genevès*, Nabil Layaïda†, Alan Schmitt‡

Project-Teams Wam and Celtique

Research Report n° 8010 — July 2012 — 15 pages

Abstract: We present a functional approach to design rich type systems based on an elegant logical representation of types. The representation is not only clean but avoids exponential increases in combined complexity due to subformula duplication. This opens the way for solving a wide range of problems such as subtyping in exponential-time even though their direct translation into the underlying logic results in an exponential blowup of the formula size, yielding an incorrectly presumed two-exponential time complexity.

Key-words: logic, lean, duplication, formula size

* CNRS
† Inria
‡ Inria

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Combinateurs logiques pour des systèmes de types riches

Résumé : Nous présentons une approche fonctionnelle pour concevoir des systèmes de types riches basée sur une représentation élégante et logique des types. La représentation n'est pas seulement propre, mais évite une augmentation exponentielle de la complexité en raison de duplication de sous-formules. Cela ouvre la voie pour résoudre un large éventail de problèmes tels que le sous-typage en temps simplement exponentiel, même si leur traduction directe dans la logique sous-jacente produit une explosion combinatoire de la taille de la formule, donnant une complexité en temps incorrectement présumée doublement exponentielle.

Mots-clés : logique, duplication, taille des formules

1 Introduction

During the last few years, a growing interest has been seen in the use of logical solvers, such as satisfiability solvers and satisfiability-modulo solvers, in the context of functional programming and static type checking [4, 3, 10]. Any progress made for enhancing the expressivity or succinctness of underlying logics may have a drastic impact on static type systems for functional programming languages. In particular, logical advances make it possible to obtain efficient procedures for deciding subtyping relations in increasingly richer type systems. For example, solvers for tree logics [9, 7] are used as basic building blocks for type systems for XQuery [10].

In this context, we generalize a logic already used for solving XPath static analysis [9] and subtyping for parametric polymorphic types [10] with combinators. These combinators allow the description of advanced type properties such as cardinality constraints, tree type frontiers, and other complex properties requiring heavy subformula duplication. We show that this approach is not only elegant but avoids exponential increases in combined complexity due to subformula duplication.

From a logical point of view, by embedding types into logical frameworks, we can capture types that are of very high expressive power, but with precise complexity bounds for reasoning. However, some advanced features in type systems such as the ones underlying XML include several constructs that cannot easily or succinctly be translated in logics as the μ-calculus. The most important of such constructs are inverse programs, nominals, and counting constraints. Inverse programs allow to navigate backwards along accessibility relations [13]. Nominals are propositional variables interpreted as singleton sets. Counting operators enable statements about the number of occurrences of nodes satisfying a given formula. Since they allow counting anywhere in trees, they can be seen as a powerful generalization of graded modalities [5, 2].

As reported by Bonati and al [5], the combination of these three features leads to undecidability for the general μ-calculus [14]. Another concern in such embeddings is related to the translation cost. Even when the problem at hand is known to be decidable, one needs to control possible blowups, via duplications, in the logical translation of problems. The logic considered here is a restricted form of μ-calculus tuned for trees which remains decidable with these three features combined, unlike [5]. More precisely, these features are described as combinators which preserve the exact complexity bound of the original logic.

Two essential steps are involved in the reduction of a problem such as subtyping to logical satisfiability: (1) the translation of the initial problem into a logical formula (using a derivor), 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 sub-formulas 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 [9, 11, 12]. The reason lies in the fact that satisfiability-testing algorithms can operate directly on a Hinttika-set-like representation of formulas composed of atomic propositions and modal subformulas. In this setting, we prove that the time complexity actually depends on the number of *distinct* atomic propositions and modal sub-formulas. 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 a logic borrowed from [9]. The logic is an alternation-free μ-calculus with converse modalities whose models are finite trees. Trees are encoded in binary, without loss of generality [6], through the "first-child" and "next-sibling" modalities, respectively noted $\langle 1 \rangle$ and $\langle 2 \rangle$ (see Figure 1).

In this setting, an elegant way of building a μ-calculus formula is to apply a combinator

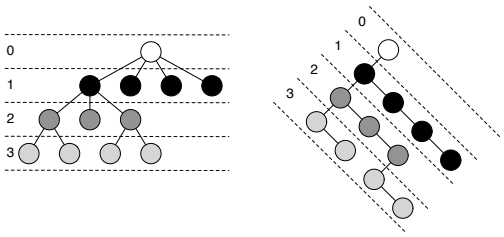

Figure 1: N-ary to Binary Tree Encoding

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. 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 recall the logic in §2, state our main result in §3, and give several examples in §4.

2 Basic Logical Formulas and Combinators

We recall the syntax of the logic of [9], 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 in Figure 1), 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. A formula is *satisfiable* if there exists a tree such that a node of this tree is *selected* by the formula. The truth formula $\top$ selects every node whereas $\bot$ selects none. The σ formula selects every node whose 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. We write $\varphi \prec \psi$ if φ is a sub-formula of ψ, and $\varphi \not\prec \psi$ if it is not.

$$\overline{\top} = \bot \qquad \overline{\sigma} = \neg\sigma$$
$$\overline{F \wedge G} = \overline{F} \vee \overline{G} \qquad \overline{\bot} = \top$$
$$\overline{\neg\sigma} = \sigma \qquad \overline{F \vee G} = \overline{F} \wedge \overline{G}$$
$$\overline{X} = \neg X \qquad \overline{x} = x$$
$$\overline{\langle a \rangle \top} = \neg\langle a \rangle \top \qquad \overline{\neg X} = X$$
$$\overline{\mu x.F} = \mu x.\overline{F} \qquad \overline{\neg \langle a \rangle \top} = \langle a \rangle \top$$
$$\overline{\langle a \rangle F \vee \neg \langle a \rangle \top} = \langle a \rangle \overline{F} \qquad \overline{\langle a \rangle F} = \langle a \rangle \overline{F} \vee \neg \langle a \rangle \top \quad F \neq \top$$

Figure 2: Negation Normal Form

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.[1] 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 defined in Figure 2. The negation of a modality is a disjunction: 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. Following [9], the greatest and smallest fixpoint coincide (for cycle-free formulas using sets of finite trees as models). Every combinator presented here is cycle-free, but 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 $\nu x.\overline{F}$.

3 Deciding Combined Formulas

3.1 The Lean

Following [11, 9, 12], we define the lean of F as follows, with $\overline{\mathcal{L}}_{\Gamma}(F)$ defined in Figure 3 (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.

$$\text{Lean}(F) = \left\{ \langle a \rangle \top \mid a \in \{1, 2, \overline{1}, \overline{2}\} \right\} \cup \{\not\epsilon\} \cup \overline{\mathcal{L}}_{\emptyset}(F)$$

3.2 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 3.1. *Let F be a combinator and φ a closed formula. We have $|Lean(F\{^{\varphi}/_{X}\})| \leq |Lean(F)| + |Lean(\varphi)|$.*

The theorem is a direct consequence of Lemma A.8 which is proved in the appendix.

We now give some intuition about this result, through a simple example. Recall the `split`(X) combinator defined as $\langle 1 \rangle X \wedge \langle 2 \rangle X$. Since the elements of the lean are either atomic propositions

[1] In case of a negated placeholder, we replace $\neg X$ with $\overline{\varphi}$, the negation normal form of φ (see Figure 2).

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

Figure 3: Negation Closed Lean

(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 $\langle 1\rangle\, \psi$).

3.3 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 [9, 11, 12]. 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 pearl is to realize 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 previously and in the following section.

4 Examples

The authors of [9] provide an online implementation [8] of their solver. Each example given in this section is provided with a boxed version which can directly be copied and pasted in order to be tested with this online implementation.

4.1 Split

The combinator $\texttt{split}(X)$ introduced in Section 3.2 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 expanded formula $\psi = \texttt{split}(\texttt{split}(\texttt{split}(\varphi)))$ uses 8 occurrences of φ. To give this formula to the implementation of [8], 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())))
```

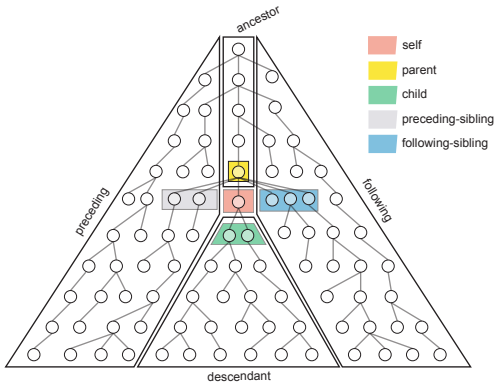

Figure 4: Tree Navigation

We then observe that ψ contains 24 atomic propositions, 38 modalities, 23 conjunctions, and 8 disjunctions (including duplicates). The size of the lean is only 18 (14 modalities and 4 atomic propositions.)[2] Each new `split(_)` around the formula then only adds two elements to the lean.

4.2 Document-Order Relation and Global Counting

We illustrate tree navigation in Figure 4. 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:

$$\texttt{following}(X) = \texttt{ancestor_or_self}(\psi)$$

where

$$\psi = \texttt{following_sibling}(\texttt{descendant_or_self}(X))$$

These combinators, whose intuition is illustrated in Figure 4, are predefined in [8]. 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);
```

[2]Notice that in their implementation, the authors of [9] do not have a lean closed under negation. Closing the lean under negation adds a modal formula $\langle a \rangle\, \overline{\varphi}$ for every modal formula $\langle a \rangle\, \varphi$ where φ is not $\top$. In this particular case, the lean would contain 10 other formulas. Each new `split(_)` would then add 4 formulas to the lean.

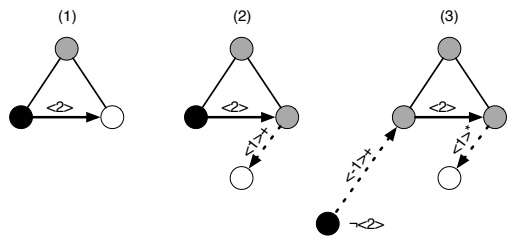

Figure 5: Tree Frontier Case Analysis

```
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() & <2>($y & leaf())) | (leaf() &
 up_until_rsibl(<2>down_to_first_leaf($y)));

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

Figure 6: Tree Frontier Example (including all the necessary definitions for use with [8]).

```
root() = ~<-1>T & ~<-2>T;
nominal($x) = ancestor_or_self(root()
                & next($x & ~next($x)));
nominal(psi())
```

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, 933 modalities, 343 conjunctions, and 590 disjunctions). However, the size of its lean is 48 (44 modalities and 4 atomic propositions).

4.3 The Tree Frontier

We borrow from [1] another advanced example: the description of properties on a tree frontier. A tree frontier is the set of leaves (nodes without a "1" child) 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. These are depicted in Figure 5, where the current leaf is black, the next leaf is white, and grayed nodes are not selected. Dotted arrows correspond to sequences of navigation.

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 in which the first member of the disjunction captures the first case and the second member of the disjunction captures the two remaining cases:

$$\texttt{next_frontier_node}(Y) = \\ (\texttt{leaf} \wedge \langle 2 \rangle\, (Y \wedge \texttt{leaf})) \vee (\texttt{leaf} \wedge \texttt{up_until_rsibl}(\psi))$$

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

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

Using these combinators, we can now express properties on the tree frontier. For instance, the formula shown on Figure 6 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 13 atomic propositions, 58 modalities, 30 variables, 13 fixpoint binders, 21 negations, 33 conjunctions, and 25 disjunctions. However, the size of the corresponding lean is 23. The lean is only composed of 21 distinct modalities and 2 distinct atomic propositions.

We remark that each additional nested call to the combinator $_ \wedge \texttt{next_frontier_node}(Y)$ extends the lean by 7 modalities. However, the corresponding global formula goes from 58 modalities to 184 for the first addition, then it goes to 562 for the second addition.

5 Conclusion

We have presented an elegant logical representation of types that can be used for designing rich type systems. The representation is not only clean but avoids exponential increases in combined complexity due to subformula duplication. Specifically, we have presented an analysis of 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 shows 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 experimentally tested this analysis with an implementation available online; it would be interesting to check whether our observation also hold experimentally with the implementations of [11] and [12]. One could also use this approach to translate problems which have been avoided as they were leading to formula duplication.

References

[1] 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.

[2] E. Bárcenas, P. Genevès, N. Layaïda, and A. 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. IJCAI/AAAI, Jul 2011.

[3] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *Proceedings of the VLDB Endowment*, 3(1):906–917, 2010.

[4] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th international conference on functional programming (ICFP '10)*, pages 105–116, Baltimore, MD, USA, 2010.

[5] P. Bonatti, C. Lutz, A. Murano, and M. Vardi. The complexity of enriched mu-calculi. *Logical Methods in Computer Science*, 4(3), 2008.

[6] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems (TACAS '08)*, pages 337–340, Budapest, 2008.

[8] P. Genevès, N. Layaïda, and A. Schmitt. XML reasoning solver project. `http://wam.inrialpes.fr/websolver/`.

[9] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI*, 2007.

[10] N. Gesbert, P. Genevès, and N. 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.

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

[12] Y. Tanabe, K. Takahashi, and M. Hagiya. A decision procedure for alternation-free modal μ–calculi. In C. Areces and R. Goldblatt, editors, *Advances in Modal Logic*, pages 341–362. College Publications, 2008.

[13] M. Y. Vardi. Reasoning about the past with two-way automata. In *ICALP*, 1998.

[14] J. Zappe. Modal Î¼-calculus and alternating tree automata. In E. GrÃ¤del, W. Thomas, and T. Wilke, editors, *Automata Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 205–211. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36387-4_10.

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

Figure 7: Number of Recursive Expansion

A Proofs

We define the *number of recursive expansions* of F or φ, written $\mathcal{E}_\emptyset(F)$, in Figure 7.

Definition A.1. *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 A.2. *We have $\overline{F\{^G/_x\}} = \overline{F}\{^{\overline{G}}/_x\}$.*

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

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

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

Lemma A.4. *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 A.5. *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 A.6. *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 $x \not\prec F$ 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.

$$\begin{aligned}
\overline{\mathcal{L}}_\Gamma(\mu x.F) &= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\}) \\
&= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(\overline{F\{^{\mu x.F}/_x\}}) && \text{by induction} \\
&= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.F\}}(\overline{F}\{^{\mu x.\overline{F}}/_x\}) && \text{by Lemma A.2} \\
&= \overline{\mathcal{L}}_{\Gamma\cup\{\mu x.\overline{F}\}}(\overline{F}\{^{\mu x.\overline{F}}/_x\}) && \text{by Lemma A.5} \\
&= \overline{\mathcal{L}}_\Gamma(\mu x.\overline{F})
\end{aligned}$$

□

Lemma A.7. *For all Γ and F, if $X \not\prec F$, 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 $X \not\prec F$ 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 A.8. *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 $X \not\prec F$. In this case, X also does not occur in $\overline{\mathcal{L}}_\Gamma(F)$.

$$\begin{aligned}
\overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\}) &= \overline{\mathcal{L}}_\Gamma(F) \\
&\subseteq \overline{\mathcal{L}}_\emptyset(F) && \text{by Lemma A.4} \\
&= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F) && \text{by Lemma A.7} \\
&= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\} \\
&\subseteq \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\} \cup \overline{\mathcal{L}}_\emptyset(\varphi)
\end{aligned}$$

Case X. We compute as follows, using Lemma A.4 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 A.4 for the set inclusion, and Lemma A.6 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.

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

Case $F \vee G$. Identical to the previous case.
Case $\langle a \rangle\, F$. We compute as follows, using Lemma A.3 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 $x \not\prec F$. We compute as follows.

$$\begin{aligned}
&\overline{\mathcal{L}}_\Gamma((\mu x.F))\{^\varphi/_X\} \\
&= \overline{\mathcal{L}}_\Gamma(\mu x.F\{^\varphi/_X\}) && \varphi \text{ closed} \\
&= \overline{\mathcal{L}}_\Gamma(F\{^\varphi/_X\}) && x \not\prec F\{^\varphi/_X\} \\
&\subseteq \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(F)\{^\varphi/_X\} \cup \overline{\mathcal{L}}_\emptyset(\varphi) && \text{by induction} \\
&= \overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(\mu x.F)\{^\varphi/_X\} \cup \overline{\mathcal{L}}_\emptyset(\varphi) && x \not\prec F
\end{aligned}$$

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 $(\Gamma \cup \{\mu x.F\{^\varphi/_X\}\})^\varphi_X = (\Gamma)^\varphi_X \cup \{\mu x.F\}$ 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 $\overline{\mathcal{L}}_{(\Gamma)^\varphi_X}(\mu x.F) = \overline{\mathcal{L}}_{(\Gamma)^\varphi_X\cup\{\mu x.F\}}(F\{^{\mu x.F}/_x\})$. We may conclude the computation.

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

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

Contents

1 Introduction **3**

2 Basic Logical Formulas and Combinators **4**

3 Deciding Combined Formulas **5**
3.1 The Lean 5
3.2 The Factorization Power of the Lean 5
3.3 Satisfiability-Testing Algorithms based on the Lean 6

4 Examples **6**
4.1 Split 6
4.2 Document-Order Relation and Global Counting 7
4.3 The Tree Frontier 8

5 Conclusion **9**

A Proofs **11**

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399