A Logical Approach To Deciding Semantic Subtyping - Supporting function, intersection, negation, and polymorphic types

Nils Gesbert, Pierre Genevès, Nabil Layaïda

A Logical Approach To Deciding Semantic Subtyping – Supporting function, intersection, negation, and polymorphic types

Nils Gesbert, Grenoble INP – Ensimag
Pierre Genevès, CNRS
Nabil Layaïda, Inria

We consider a type algebra equipped with recursive, product, function, intersection, union, and complement types together with type variables and implicit universal quantification over them. We consider the subtyping relation recently defined by Castagna and Xu over such type expressions and show how this relation can be decided in EXPTIME, answering an open question. The novelty, originality and strength of our solution reside in introducing a logical modeling for the semantic subtyping framework. We model semantic subtyping in a tree logic and use a satisfiability-testing algorithm in order to decide subtyping. We report on practical experiments made with a full implementation of the system. This provides a powerful polymorphic type system aiming at maintaining full static type-safety of functional programs that manipulate trees, even with higher-order functions, which is particularly useful in the context of XML.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Algorithms, Design, Languages, Theory, Verification

Additional Key Words and Phrases: Type-system, Polymorphism, subtyping

ACM Reference Format:
Nils Gesbert, Pierre Genevès, Nabil Layaïda, 2013. A Logical Approach To Deciding Semantic Subtyping. X, X, Article XX (X XXXX), 26 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In programming, subtyping represents a notion of safe substitutability: τ being a subtype of τ' means that wherever in the program something of type τ' is used, it is safe to use something of type τ instead. This property has a natural set-theoretic interpretation: the set of values which can safely replace something of type τ is included in the set of values which can safely replace something of type τ'.

The semantic subtyping approach consists of using this set-theoretic property to *define* the subtyping relation, rather than for example an axiomatic definition. Types are given an interpretation as sets and subtyping is defined as inclusion of interpretations.

The XML-centric functional language XDuce [Hosoya and Pierce 2003] uses this semantic approach to define the subtyping relation between datatypes. Datatypes in that language are intended to correspond to XML document types (as described for example by DTDs), i. e. regular tree grammars, and are built using pair construction, union, intersection, negation and recursion. The set-theoretic interpretation of a type is the regular language of trees it describes, so subtyping is inclusion of regular languages. XDuce however does not have higher-order functions, and the type system does not include functional types.

The XDuce type system was extended to include arrow types in the language CDuce [Benzaken et al. 2003]. In that language, boolean combinations of types can still be used, and intersections of arrow types are interpreted as the type of overloaded functions (which give a result of a different type depending on the type of their argument). Extending the set-theoretic interpretation of types accordingly, so that subtyping still corresponds to inclusion of interpretations, turns out to be non-trivial and the recipe

This work was supported by the ANR project TYPEX, ANR-11-BS02-007.

for managing it is explained in [Frisch et al. 2008] — we summarise it, with a slightly different focus than the original paper, in Section 2.

More recently, extending the XDuce type algebra with type variables so as to support prenex parametric polymorphism, while keeping the semantic subtyping approach, has been studied in [Hosoya et al. 2009]. Again, doing the same in the presence of arrow types was more difficult, and a solution has only been proposed in 2011 by Castagna and Xu [Castagna and Xu 2011].

In both [Frisch et al. 2008] and [Castagna and Xu 2011], algorithms used to decide the subtyping relations rely on arrow elimination. It is well known that in a sensible subtyping relation, $\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2'$ is equivalent to the conjunction of $\tau_1' \leqslant \tau_1$ and $\tau_2 \leqslant \tau_2'$, so that a subtyping decision problem involving arrows can be reduced to two problems not involving them. It gets more complicated than this example when things like intersections of arrow types are allowed but can still be done. In general, very schematically, the way the algorithms work is by splitting complex types into components and distinguishing cases repeatedly in order to reduce ultimately the problem to a series of elementary comparisons between base types. Because of that, adding new constructs to the type algebra mechanically complicates the algorithm: for example, the algorithm of [Castagna and Xu 2011] behaves like the one of [Frisch et al. 2008] for monomorphic types, but contains new rules for variable elimination in various cases depending of where they occur in the type. These additions were not easy to define and obscure the algorithm enough that proving that it terminated in all cases was difficult — it was in fact yet unproven when we first implemented the decision procedure we present here — and that its complexity is still unknown.

An interesting thing to note in these founding works about semantic subtyping is that, while the set-theoretic interpretation of types is used to give some insight and some theoretical backing to the subtyping relation, it does not play as fundamental a role as we may think in the practical applications — one does not need the semantic subtyping theoretical development to use or even to understand the relation $\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2' \Leftrightarrow \tau_1' \leqslant \tau_1 \wedge \tau_2 \leqslant \tau_2'$, after all, and the algorithm relies mostly on such transformations. The authors actually present the proof that a model of types can effectively be constructed as a way to justify that the subtyping relation makes sense as it is, which is nice to have but would not really be absolutely necessary; something almost cosmetic.

In the present paper, we show in some sense how to push the semantic approach further, all the way into the decision algorithm — we could say we present a semantic approach to deciding semantic subtyping. We give the set-theoretic model of types a practical use: types are translated into logical formulas describing precisely the set of model elements corresponding to the type. A type being a subtype of another then corresponds to the logical implication of the corresponding formulas being valid. We show that domain elements can be represented by finite trees and that the formulas corresponding to types can be written in a μ-calculus of finite trees for which we have an efficient satisfiability checker. Deciding subtyping between two types can then be done by feeding to this checker the negation of the implication formula relating the two types — if this formula is unsatisfiable, the implication is valid and thus subtyping holds; otherwise, we can exhibit explicitly a domain element which disproves the implication, that is, which belongs to the first type but not the second.

A benefit of this fully logical approach is made clear in Section 5 where we show that extending the type algebra of [Frisch et al. 2008] with type variables and altering the subtyping relation accordingly, in the way described by [Castagna and Xu 2011], can be done in a very simple way and at effectively zero cost in our system. This in turn immediately proves that subtyping is still decidable in the extended framework of [Castagna and Xu 2011] (this was chronologically, by a few days, the first satisfia-

bility proof for that relation), and furthermore gives a precise complexity bound for its decision, since the translation into logic is linear and the complexity of the solver is known — this complexity bound is one of our contributions, since no other proof of it currently exists.

1.1. The Need for Polymorphism and Subtyping: a Concrete Example

This work is motivated by a growing need for polymorphic type systems for programming languages that manipulate XML data. For instance, XQuery [Boag et al. 2007] is the standard query and functional language designed for querying collections of XML data. The support of higher-order functions, currently missing from XQuery, appears in the requirements for the forthcoming XQuery 3.0 language [Engovatov and Robie 2010]. This results in an increasing demand in algorithms for proving or disproving statements with polymorphic types, and with types of higher-order functions (like the traditional `map` and `fold` functions), or more generally, statements involving the subtyping relation over a type algebra with recursive, product, function, intersection, union, and complement types together with type variables and universal quantification over them.

For example, let us consider a simple property relating polymorphic types of functions that manipulate lists. We consider a type α, and denote by $[\alpha]$ the type of α-lists (lists whose elements are of type α). The type τ of functions that process an α-list and return a boolean is written as follows:

$$\tau = \forall\alpha.\ [\alpha] \to \texttt{Bool}$$

where $\texttt{Bool} = \{\texttt{true}, \texttt{false}\}$ is the type containing only the two values `true` and `false` Now let us consider functions that distinguish α-lists of even length from α-lists of odd length: such a function returns `true` for lists with an even number of elements of type α, and returns `false` for lists with an odd number of elements of type α. One may represent the set of these functions by a type τ' written as follows:

$$\forall\alpha.\ \text{even}[\alpha] \to \{\texttt{true}\} \wedge \text{odd}[\alpha] \to \{\texttt{false}\}$$

where `{true}` and `{false}` are singleton types (containing just one value). If we make explicit the parametric types even$[\alpha]$ and odd$[\alpha]$, τ' becomes:

$$\tau' = \forall\alpha. \begin{pmatrix} & \mu v.(\alpha \times (\alpha \times v)) \vee \text{nil} & \to \{\texttt{true}\} \\ \wedge & \mu v.(\alpha \times (\alpha \times v)) \vee (\alpha \times \text{nil}) & \to \{\texttt{false}\} \end{pmatrix}$$

where $\times$ denotes the cartesian product, μ binds the variable v for denoting a recursive type, and nil is a singleton type. Obviously, a particular function of type τ' can also be seen as a less-specific function of type τ. In other terms, from a practical point of view, a function of type τ can be replaced by a more specific function of type τ' while preserving type-safety (however the converse is not true). This is exactly what captures the notion of subtyping; in that case we write:

$$\tau' \leqslant \tau \tag{1}$$

where $\leqslant$ denotes the subtyping relation which is under scrutiny in this article.

1.2. Semantic Subtyping with Logical Solvers

During the last few years, a growing interest has been seen in the use of logical solvers such as satisfiability-testing solvers and satisfiability-modulo solvers in the context of functional programming and static type checking [Bierman et al. 2010; Benedikt and Cheney 2010]. In particular, solvers for tree logics [Genevès et al. 2007; de Moura and Bjørner 2008] are used as basic building blocks for type systems for XQuery.

The main idea in this paper is a type-checking algorithm for polymorphic types based on deciding subtyping through a logical solver. To decide whether τ is a subtype of type τ', we first construct equivalent logical formulas φ_τ and $\varphi_{\tau'}$ and then check the validity of the formula $\psi = \varphi_\tau \Rightarrow \varphi_{\tau'}$ by testing the unsatisfiability of $\neg\psi$ using the satisfiability-testing solver. This technique corresponds to semantic subtyping [Frisch et al. 2008] since the underlying logic is inherently tied to a set-theoretic interpretation. Semantic subtyping has been applied to a wide variety of types including refinement types [Bierman et al. 2010] and types for XML such as regular tree types [Hosoya et al. 2005], function types [Benzaken et al. 2003], and XPath [Clark and DeRose 1999] expressions [Genevès et al. 2007].

This fruitful connection between logics, their decision procedures, and programming languages permitted to equip the latter with rich type systems for sophisticated programming constructs such as expressive pattern-matching and querying techniques. The potential benefits of this interconnection crucially depend on the expressivity of the underlying logics. Therefore, there is an increasing demand for more and more expressiveness. For example, in the context of XML:

— SMT solvers like [de Moura and Bjørner 2008] offer an expressive power that corresponds to a fragment of first-order logic in order to solve the intersection problem between two queries [Benedikt and Cheney 2010];
— Full first-order logic solvers over finite trees [Genevès et al. 2007] solve containment and equivalence of XPath expressions;
— Monadic second-order logic solvers over trees, and – equivalent yet much more effective – satisfiability-solvers for μ-calculus over trees [Genevès et al. 2007] are used to solve query containment problems in the presence of type constraints.

1.3. Contributions of the Paper

The novelty of our work is threefold. It is the first work that:

— Proves the decidability of semantic subtyping for polymorphic types with function, product, intersection, union, and complement types, as defined by Castagna and Xu [Castagna and Xu 2011], and gives a precise complexity upper-bound: $2^{(n)}$, where n is the size of types being checked. Decidability was only conjectured by Castagna and Xu before our result, although they have now proved it independently; our result on complexity is still the only one. In addition, we provide an effective implementation of the decision procedure.
— Produces counterexamples whenever subtyping does not hold with polymorphic and arrow types. These counterexamples are valuable for programmers as they represent evidence that the relation does not hold.
— Pushes the integration between programming languages and logical solvers to a very high level. The logic in use is not only capable of ranging over higher order functions, but it is also capable of expressing values from semantic domains that correspond to monadic second-order logic such as XML tree types [Genevès et al. 2007]. This shows that such solvers can become the core of XML-centric functional languages type-checkers such as those used in CDuce [Benzaken et al. 2003] or XDuce [Hosoya and Pierce 2003].

1.4. Structure of the Paper

We introduce the semantic subtyping framework in Section 2 where we start with the monomorphic type algebra (without type variables). We present the tree logic in which we model semantic subtyping in Section 3. We detail the logical encoding of types in Section 4. Then, in Section 5 we extend the type algebra with type variables, and state the main result of the paper: we show how to decide the subtyping relation for

the polymorphic case in exponential time. We report on practical experiments using the implementation in Section 6. Finally, we discuss related work in Section 7 before concluding in Section 8.

2. SEMANTIC SUBTYPING FRAMEWORK

In this section, we present the type algebra we consider: we introduce its syntax and define its semantics using a set-theoretic interpretation. This framework is the one described at length in [Frisch et al. 2008]; we summarise its main features and give the intuitions behind it, using a slightly different point of view than the original paper, but we refer the reader to that paper for technical details.

We will then extend this framework with type variables in Section 5.

2.1. Types

Type terms are defined using the following grammar:

$$
\begin{array}{rll}
\tau ::= & & \\
 & b & \text{basic type} \\
| & \tau \times \tau & \text{product type} \\
| & \tau \to \tau & \text{function type} \\
| & \tau \vee \tau & \text{union type} \\
| & \neg\tau & \text{complement type} \\
| & \mathbf{0} & \text{empty type} \\
| & v & \text{recursion variable} \\
| & \mu v.\tau & \text{recursive type}
\end{array}
$$

We consider μ as a binder and define the notions of free and bound variables and closed terms as standard. A type is a closed type term which is *well-formed* in the sense that:

— The negation operator only occurs in front of *closed* types;
— Both operands of an arrow constructor must be closed types as well;
— Every occurrence of a recursion variable is separated from its binder by at least one occurrence of the product or arrow constructor (guarded recursion).

So, for example, $\mu v.\mathbf{0} \vee v$ is not well-formed, nor is $\mu v.\mathbf{0} \to \neg v$.

Additionally, the following abbreviations are defined:

$$\tau_1 \wedge \tau_2 \stackrel{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2)$$

and

$$\mathbf{1} = \neg\mathbf{0}$$

2.2. Set-theoretic interpretation

2.2.1. Underlying ideas. Before defining formally how types shall be interpreted, let us summarise the ideas which lead to that interpretation.

Consider a programming language whose values are constants from a set $\mathcal{C}$, pairs of values, and functions. We consider the different kinds of values disjoint, i. e. for example no value can simultaneously be a pair and a function. Let $\mathcal{W}$ be the set of all values in the language. The basic idea of the semantic subtyping framework is to interpret the types of the above algebra as subsets of $\mathcal{W}$, giving $\vee$ and $\neg$ the meaning of set-theoretic union and complement, and to define subtyping as set inclusion of interpretations.

Suppose we have an interpretation of base types b as sets of constants. As long as we don't use arrows, it is straightforward to define a set-theoretic semantics for $\times$. The recursive type $\mu v.\tau$ can be interpreted as a least fixpoint.

The usual interpretation of a function type however is operational rather than set-theoretic. Indeed, we can consider in the general case that when applying a function to an argument, a computation is triggered which can, possibly nondeterministically, either yield a value, yield an error or yield nothing (i. e. not terminate). The intended meaning of the type $\tau_1 \to \tau_2$ is that whenever applied to an argument of type τ_1, the function returns either a value of type τ_2 or nothing. An important feature of this framework is that it allows overloaded functions: a function f can return something of type τ_2 when given an argument of type τ_1, and return something of the completely different type τ_3 when given an argument of type τ_4. In that case, f has *both* type $\tau_1 \to \tau_2$ and type $\tau_4 \to \tau_3$, and since the type algebra allows boolean combinations of types, it also has type $\tau_1 \to \tau_2 \wedge \tau_4 \to \tau_3$, which is more precise than each simple arrow type.

This operational definition of arrow types makes impractical to interpret them as sets of actual function values defined in the considered language. Rather, [Frisch et al. 2008] propose to use the associated abstract functions, i. e. sets of pairs of an antecedent and a result. Note that because the computational functions are allowed to be nondeterministic, the abstract ones are not necessarily functions in the mathematical sense but more general relations. Formally, an abstract function is a subset of $\mathcal{W} \times (\mathcal{W} \cup \{\Omega\})$. Each pair (d, d') in the set means that, when given d as an argument, the function *may* yield d' as a result. If d does not appear as the first element of any pair, the operational interpretation is that the function can still accept d as an argument but will not yield a result: this represents a computation which does not terminate. A pair of the form (d, Ω) is used to represent a function rejecting d as an argument: when given d, it yields an error.

The set of abstract functions of type $\tau_1 \to \tau_2$ can then be defined simply as all sets of pairs (d, d') such that whenever d is of type τ_1, d' is of type τ_2. This is called the extensional interpretation of function types in [Frisch et al. 2008]. Formally:

$$\mathbb{E}[\![\tau_1 \to \tau_2]\!] \stackrel{\text{def}}{=} \{S \subseteq \mathcal{W} \times (\mathcal{W} \cup \{\Omega\}) \mid (d, d') \in S \wedge (d : \tau_1) \Rightarrow (d : \tau_2)\}$$

where $(d : \tau)$ means the value d has type τ. Boolean combinators can be interpreted as the corresponding set-theoretic operations on extensional interpretations, and subtyping corresponds to inclusion between sets of abstract functions.

This extensional interpretation has the problem that not all abstract functions can have concrete implementations in the language, for cardinality reasons: the set of concrete functions is included in $\mathcal{W}$ since they are values themselves, but the set of all possible abstract functions is $\mathcal{P}(\mathcal{W} \times (\mathcal{W} \cup \{\Omega\}))$. However, inclusion between the extensional interpretations of two types clearly implies inclusion between the sets of values of those types, and for the converse implication to hold, it suffices that every type whose extensional interpretation is non-empty has a witness in the language. Indeed, because we have boolean combinators in the type algebra, the question of inclusion reduces to a question of emptiness.

It is not immediately obvious that a language with that property (i. e. that whenever there exists an abstract function of some type, there is also a function of that type in the language) exists. However the following property makes it easy to define one: whenever there exists an abstract function of some type, there also exists a **finite** abstract function (i. e. the set of pairs is finite) of the same type. To get an intuition of why this is true, remark that for an abstract function to have type $\tau_1 \to \tau_2$, it suffices

---

The attentive reader may remark that the complement of an arrow type includes not just all functions which do not have that type but also all non-functional values. In the full formal development in [Frisch et al. 2008], the extensional interpretation of a type is actually a subset of the disjoint union of non-functional values and abstract functions, so that this kind of things is taken into account.

that it contains *no* pair (a, b) with $(a : \tau_1)$ and $(b : \neg\tau_2)$. For it to have type $\neg(\tau_1 \to \tau_2)$, it suffices that it contains *one* such pair. Since the type algebra only allows ***finite*** boolean combinations of types, it is quite clearly impossible to build a type constraint that would be satisfied only by infinite sets of pairs.

Therefore, if we consider an abstract language where function values are simply finite lists of pairs of values, with the semantics described above, the semantic subtyping relation it induces on types is the same as any sufficiently expressive concrete language with the same set of base constants. We now define formally our semantic domain.

2.2.2. Formal definitions. Consider an arbitrary set $\mathcal{C}$ of constants. From it, we define the semantic domain $\mathcal{D}$ as the set of ds generated by the following grammar, where c ranges over constants in $\mathcal{C}$:

d	::=		domain element
		c	base constant
	|	(d, d)	pair
	|	$\{(d, d'), \ldots, (d, d')\}$	function
d'	::=		extended domain element
		d	
	|	Ω	error

We suppose we have an interpretation $\mathbb{B}[\![\cdot]\!]$ of basic types b as subsets of $\mathcal{C}$.

To properly define the typing relation between extended domain elements and types, we first define a structural ordering relation $\trianglelefteq$ on $(\mathcal{D} \cup \{\Omega\}) \times T$ where T is the set of types:

— On extended domain elements, we use the ordering $d'_1 \trianglelefteq d'_2$ if d'_1 is a subterm of d'_2;
— Let the *shallow depth* of a type term be the longest path, in its syntactic tree, starting from the root and consisting only of μ, $\vee$, and $\neg$ nodes. We order types by $\tau_1 \trianglelefteq \tau_2$ if the shallow depth of τ_1 is less than the shallow depth of τ_2;
— Pairs are ordered lexicographically, i. e. $(d'_1, \tau_1) \trianglelefteq (d'_2, \tau_2)$ if either $d'_1 \triangleleft d'_2$ or $d'_1 = d'_2$ and $\tau_1 \trianglelefteq \tau_2$.

Recall the well-formedness constraint on types : in the syntactic tree, a recursion variable is always separated from its binder by a $\times$ or $\to$ constructor. This implies that the unfolding of a recursive type always has a strictly smaller shallow depth than the original type: $\mu v.\tau \triangleright \tau\{^{\mu v.\tau}/_v\}$; indeed, the substitution may increase the depth of the syntactic tree, but only below a $\times$ of $\to$ node, so it does not affect its shallow depth.

The predicate $(d' : \tau)$ where d' is either an element of $\mathcal{D}$ or Ω and τ is a type can now be defined recursively in the following way:

$$
\begin{aligned}
(\Omega : \tau) &= \mathsf{false} \\
(c : b) &= c \in \mathbb{B}[\![b]\!] \\
((d_1, d_2) : \tau_1 \times \tau_2) &= (d_1 : \tau_1) \wedge (d_2 : \tau_2) \\
(\{(d_1, d'_1), \ldots, (d_n, d'_n)\} : \tau_1 \to \tau_2) &= \forall i, (d_i : \tau_1) \Rightarrow (d'_i : \tau_2) \\
(d : \tau_1 \vee \tau_2) &= (d : \tau_1) \vee (d : \tau_2) \\
(d : \neg\tau) &= \neg(d : \tau) \\
(d : \mu v.\tau) &= (d : \tau\{^{\mu v.\tau}/_v\}) \\
(d : \tau) &= \mathsf{false} \text{ in any other case}
\end{aligned}
$$

It is easy to check that all occurrences of the predicate on the right-hand side of the definition are for pairs strictly smaller, with respect to $\trianglelefteq$, than the one on the left. Because all terms and types are finite, this makes the definition well-founded.

The interpretation of types as parts of $\mathcal{D}$ is then defined as $[\![\tau]\!] = \{d \mid (d : \tau)\}$. Note that Ω is not part of any type, as expected.

In this framework, we consider XML types as regular tree languages. An XML tree type is interpreted as the set of documents that match the type.

Finally, the subtyping relation is defined as $\tau_1 \leqslant \tau_2 \Leftrightarrow [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$, or, equivalently, $[\![\tau_1 \wedge \neg\tau_2]\!] = \emptyset$.

3. TREE LOGIC FRAMEWORK

In this section we introduce the logic in which we model the semantic subtyping framework. This logic is a subset of the one proposed in [Genevès et al. 2007]: a variant of μ-calculus whose models are finite trees. We first introduce below the syntax and semantics of the logic, before tuning it for representing types.

3.1. Formulas

Formulas are defined thus:

φ, ψ	$::=$		formula
		$\top$	true
	$\mid$	$\sigma \mid \neg\sigma$	atomic proposition (negated)
	$\mid$	X	variable
	$\mid$	$\varphi \vee \psi$	disjunction
	$\mid$	$\varphi \wedge \psi$	conjunction
	$\mid$	$\langle a \rangle \varphi \mid \neg \langle a \rangle \top$	existential (negated)
	$\mid$	$\mu(X_i = \varphi_i)_{i \in I}$ in ψ	(least) n-ary fixpoint

where $a \in \{1, 2\}$ are *programs*, and I is a finite set. Atomic propositions σ correspond to labels from a countable set Σ. Additionnally, we use the abbreviation $\mu X.\varphi$ for $\mu(X = \varphi)$ in φ.

Intuitively, the logic allows one to formulate regular properties on unranked trees: the programs "1" and "2" are respectively used to access the first child node and the next sibling node in an unranked tree. For instance, the formula $a \wedge \langle 1 \rangle (b \wedge \langle 2 \rangle c)$ is satisfied at the root of the tree denoted by the term $a(b, c)$. The recursive formula $\langle 1 \rangle (\mu X.a \vee \langle 1 \rangle X \vee \langle 2 \rangle X)$ states the existence of some node labelled with "a" at an arbitrary depth in the subtree. We formalize those intuitions in the next sections.

3.2. Semantic domain

The semantic domain is the set $\mathcal{F}$ of focused trees defined by the following syntax, where we have an alphabet Σ of labels, ranged over by σ:

t	$::=$	$\sigma[tl]$	tree
tl	$::=$		list of trees
		ϵ	empty list
	$\mid$	$t :: tl$	cons cell
c	$::=$		context
		(tl, Top, tl)	root of the tree
	$\mid$	$(tl, c[\sigma], tl)$	context node
f	$::=$	(t, c)	focused tree

A focused tree (t, c) is a pair consisting of a tree t and its context c. The context $(tl, c[\sigma], tl)$ comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the

current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree may be Top if the current tree is at the root, otherwise it is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

The *name* of a focused tree is defined as $\mathtt{nm}(\sigma[tl], c) = \sigma$.

We now describe how to navigate focused trees, in binary style. There are four directions, or *modalities*, that can be followed: for a focused tree f, $f\langle 1\rangle$ changes the focus to the first child of the current tree, $f\langle 2\rangle$ changes the focus to the next sibling of the current tree, $f\langle \overline{1}\rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\langle \overline{2}\rangle$ changes the focus to the previous sibling.

Formally, we have:

$$(\sigma[t :: tl], c)\langle 1\rangle \stackrel{\text{def}}{=} (t, (\epsilon, c[\sigma], tl))$$
$$(t, (tl_l, c[\sigma], t' :: tl_r))\langle 2\rangle \stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma], tl_r))$$
$$(t, (\epsilon, c[\sigma], tl))\langle \overline{1}\rangle \stackrel{\text{def}}{=} (\sigma[t :: tl], c)$$
$$(t', (t :: tl_l, c[\sigma], tl_r))\langle \overline{2}\rangle \stackrel{\text{def}}{=} (t, (tl_l, c[\sigma], t' :: tl_r))$$

When the focused tree does not have the required shape, these operations are not defined.

3.3. Interpretation

Formulas are interpreted as subsets of $\mathcal{F}$ in the following way, where V is a mapping from variables to formulas:

$$[\![\top]\!]_V \stackrel{\text{def}}{=} \mathcal{F} \qquad [\![\sigma]\!]_V \stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) = \sigma\}$$
$$[\![X]\!]_V \stackrel{\text{def}}{=} V(X) \qquad [\![\neg\sigma]\!]_V \stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) \neq \sigma\}$$
$$[\![\varphi \vee \psi]\!]_V \stackrel{\text{def}}{=} [\![\varphi]\!]_V \cup [\![\psi]\!]_V \qquad [\![\varphi \wedge \psi]\!]_V \stackrel{\text{def}}{=} [\![\varphi]\!]_V \cap [\![\psi]\!]_V$$

$$[\![\langle a\rangle \varphi]\!]_V \stackrel{\text{def}}{=} \{f\langle \overline{a}\rangle \mid f \in [\![\varphi]\!]_V \wedge f\langle \overline{a}\rangle \text{ defined}\}$$
$$[\![\neg\langle a\rangle \top]\!]_V \stackrel{\text{def}}{=} \{f \mid f\langle a\rangle \text{ undefined}\}$$

$$[\![\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=}$$
$$\text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, [\![\varphi_j]\!]_{V[\overline{T_i/X_i}]} \subseteq T_j\} \quad \text{in}$$
$$\text{let } (U_j) = \left(\textstyle\bigcap_{(T_i)\in S} T_j\right)_{j\in I} \quad \text{in} \quad [\![\psi]\!]_{V[\overline{U_i/X_i}]}$$

where $V[\overline{T_i/X_i}](X) = V(X)$ if $X \notin \{X_i\}$ and T_i if $X = X_i$.

The lemma 4.2 of [Genevès et al. 2007] says that the interpretation of a fixpoint formula is equal to the union of the interpretations of all its finite unfoldings (where unfolding is defined as usual). A consequence (detailed in [Genevès et al. 2007]) is that the logic is closed under negation, i. e. for any closed φ, $\neg\varphi$ can be expressed in the syntax using De Morgan's relations and this definition:

$$\neg\langle a\rangle \varphi \stackrel{\text{def}}{=} \neg\langle a\rangle \top \vee \langle a\rangle \neg\varphi$$
$$\neg\mu(X_i = \varphi_i) \text{ in } \psi \stackrel{\text{def}}{=} \mu(X_i = \neg\varphi_i\{\overline{X_i/\neg X_i}\}) \text{ in } \neg\psi\{\overline{X_i/\neg X_i}\}$$

In the following, we consider only closed formulas and write $[\![\varphi]\!]$ for $[\![\varphi]\!]_\emptyset$.

4. LOGICAL ENCODING

In the context of the present paper, we want finite tree models of the logic to correspond to types introduced in section 2. Thus, we first extend the alphabet of node labels to be

able to reason with type constructors. Then, we present the translation of a type into a logical formula.

4.1. Representation of domain elements

Let $\mathcal{T}$ be the set of (unfocused) trees. Set $\mathcal{C} = \{\mathbb{B}[tl] \mid tl \in \mathcal{T}^*\}$, where $\mathbb{B}$ is a label **not in** Σ : the set of trees with a distinguished root $\mathbb{B}$. Let $\mathcal{T}_{ext}$ be the set of trees obtained by extending Σ with the four extra labels $(\rightarrow), (\times), \mathbb{B}$ and Ω. Then $\mathcal{D}_\Omega$ can straightforwardly be embedded into $\mathcal{T}_{ext}$ in the following way:

$$\begin{aligned}
\mathsf{tree}(c) &= c \\
\mathsf{tree}(\Omega) &= \Omega[\epsilon] \\
\mathsf{tree}(d, d') &= (\times)[\mathsf{tree}(d) :: \mathsf{tree}(d') :: \epsilon] \\
\mathsf{tree}(\{(d_1, d'_1), \ldots, (d_n, d'_n)\}) &= (\rightarrow)[\mathsf{tree}(d_1, d'_1) :: \ldots :: \mathsf{tree}(d_n, d'_n) :: \epsilon]
\end{aligned}$$

The intuition of this tree representation is illustrated in Figure 1.

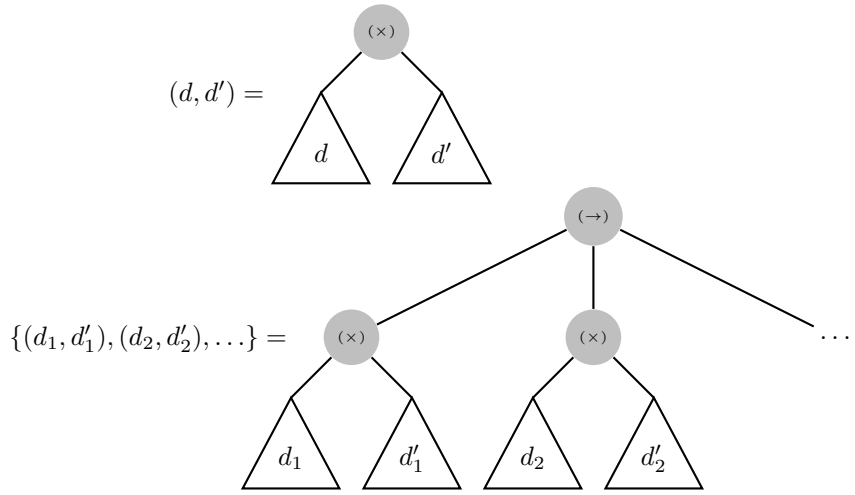

Fig. 1. Pairs and functions are represented as trees with special labels.

In the following we consider this embedding implicitly done, so $\mathcal{D}_\Omega \subseteq \mathcal{T}_{ext}$.

4.2. Translation of types

First of all, we can define basic types b, which are to represent sets of trees with no special nodes but the distinguished root $\mathbb{B}$, as the (closed) base formulas of the logic. The full interpretation of formulas uses sets of focused trees, but note that a toplevel formula cannot contain any constraint on what is above or to the left of the node at focus, so it can be considered as describing just a list of trees. The interpretation of a base type will then be a $\mathbb{B}$ root whose list of children is described by the formula. Formally:

$$\mathbb{B}[\![\varphi]\!] \stackrel{\text{def}}{=} \{\mathbb{B}[t :: tl_2] \mid (t, (tl_1, c[\sigma], tl_2)) \in [\![\varphi]\!]\}$$

Note how the only part of the context taken into account in defining the semantics is the list of following siblings of the current node.

Then, we translate the types into *extended* formulas obtained (as for extended trees) by adding to Σ the labels $(\times), (\to), \Omega$ and $\mathbb{B}$. Straightforwardly these formulas denote lists of trees in $\mathcal{T}_{ext}$.

First, we define the following formulas:

$$\begin{aligned}
\mathsf{isbase} &= \mu X.((\neg\langle 1\rangle\top \vee \langle 1\rangle X) \wedge (\neg\langle 2\rangle\top \vee \langle 2\rangle X)\\
&\qquad \wedge \neg\mathbb{B} \wedge \neg(\to) \wedge \neg(\times) \wedge \neg\Omega)\\
\mathsf{error} &= \Omega \wedge \neg\langle 1\rangle\top\\
\mathsf{isd} &= \mu X.(\\
&\qquad (\mathbb{B} \wedge \langle 1\rangle \mathsf{isbase}) \vee\\
&\qquad ((\times) \wedge \langle 1\rangle (X \wedge \langle 2\rangle (X \wedge \neg\langle 2\rangle\top))) \vee\\
&\qquad ((\to) \wedge (\neg\langle 1\rangle\top \vee\\
&\qquad\quad \langle 1\rangle \mu Y.((\neg\langle 2\rangle\top \vee \langle 2\rangle Y) \wedge\\
&\qquad\qquad (\times) \wedge \langle 1\rangle (X \wedge \langle 2\rangle ((X \vee \mathsf{error}) \wedge \neg\langle 2\rangle\top))\\
&\qquad))))
\end{aligned}$$

isbase selects all tree lists which do not contain any of the special labels (the fixpoint is for selecting all the nodes). error is straightforward. isd selects all elements of $\mathcal{D}$ (actually, all tree lists whose first element is in $\mathcal{D}$): either they are a constant (a $\mathbb{B}$ node with a base list as children), or a pair (a $(\times)$ node with exactly two children each of which is itself in $\mathcal{D}$), or a function: a $(\to)$ node with either no children at all or a list of children (described by Y) all of which are pairs whose second element may be error.

We now associate to every type τ the formula $\mathsf{fullform}(\tau) = \mathsf{isd} \wedge \mathsf{form}(\tau)$, with $\mathsf{form}(\tau)$ defined as follows, where X_v is a different variable for every v and is also different from X:

$$\begin{aligned}
\mathsf{form}(b) &= \mathbb{B} \wedge \langle 1\rangle b\\
\mathsf{form}(\tau_1 \times \tau_2) &= (\times) \wedge \langle 1\rangle (\mathsf{form}(\tau_1) \wedge \langle 2\rangle \mathsf{form}(\tau_2))\\
\mathsf{form}(\tau_1 \to \tau_2) &= (\to) \wedge (\neg\langle 1\rangle\top \vee\\
&\qquad \langle 1\rangle \mu X.((\neg\langle 2\rangle\top \vee \langle 2\rangle X)\\
&\qquad\quad \wedge \langle 1\rangle (\neg\mathsf{form}(\tau_1) \vee \langle 2\rangle \mathsf{form}(\tau_2)))\\
&\qquad)\\
\mathsf{form}(\tau_1 \vee \tau_2) &= \mathsf{form}(\tau_1) \vee \mathsf{form}(\tau_2)\\
\mathsf{form}(\neg\tau) &= \neg\mathsf{form}(\tau)\\
\mathsf{form}(\mathbf{0}) &= \neg\top\\
\mathsf{form}(v) &= X_v\\
\mathsf{form}(\mu v.\tau) &= \mu X_v.\mathsf{form}(\tau)
\end{aligned}$$

Recall that basic types b are themselves formulas, but that their interpretation as a type is different from their interpretation as a formula (see the first paragraph of Section 4.2 and the definition of $\mathbb{B}[\![\varphi]\!]$, the interpretation as a type, in terms of $[\![\varphi]\!]$, the interpretation as a formula). This explains why the translation of b contains b itself. The translation of product types is simple: it describes a $(\times)$ node whose first child is described by $\mathsf{form}(\tau_1)$ and has a following sibling described by $\mathsf{form}(\tau_2)$. The translation of arrow types has a structure similar to what appeared in isd: it describes a $(\to)$ node with either no children or a list of children recursively described by X (each node has either no following sibling or a following sibling itself described by X). Each of these nodes must have a first child which either is not of type τ_1 or has a next sibling of type

τ_2 — this means that these nodes represent pairs (d_i, d'_i) such that $(d_i : \tau_1) \Rightarrow (d'_i : \tau_2)$. The attentive reader may notice that the formula $\mathsf{form}(\tau_1 \to \tau_2)$ does not enforce in itself that all children of the $(\to)$ node are actually pairs; the reason for that is that isd already enforces it.

We can see that the formulas in the translation do not contain any $\langle 2 \rangle$ at toplevel (i. e. not under $\langle 1 \rangle$), nor does isd. This means they describe a single tree (they say nothing on its siblings), or in other words that in their interpretation as focused trees, the context is completely arbitrary, as it is not constrained in any way. Formally, we thus define the restricted interpretation of extended formulas as follows:

$$\mathbb{F}[\![\varphi]\!] \stackrel{\text{def}}{=} \{t \mid (t, c) \in [\![\varphi]\!]\}$$

That is, we drop the context completely.

Then we have $\mathbb{F}[\![\mathsf{fullform}(\tau)]\!] = [\![\tau]\!]$. This is a particular case of the property for polymorphic types which will be proved in the following section.

The main consequence of this property is that a type τ is empty if and only if the interpretation of the corresponding formula is empty — which is equivalent to the formula being unsatisfiable. Because there exists a satisfiability-checking algorithm for this tree logic [Genevès et al. 2007], this means that this translation gives an alternative way to decide the classical semantic subtyping relation as defined in [Frisch et al. 2008]. More interestingly, it yields a decision procedure for the subtyping relation *in the polymorphic case as well*, as we will explain in the next section.

5. POLYMORPHISM: SUPPORTING TYPE VARIABLES

So far we have described a new, logic-based approach to a question — semantic subtyping in the presence of intersection, negation and arrow types — which had already been studied. We now show how this new approach allows us, in a very natural way, to encompass the latest work by adding polymorphism to the types along the lines of [Castagna and Xu 2011].

We add to the syntax of types *variables*, α, β, γ taken from a countable set $\mathcal{V}$. If τ is a polymorphic type, we write $\mathsf{var}(\tau)$ the set of variables it contains and call *ground type* a type with no variable. We sometimes write $\tau(\overline{\alpha})$ to indicate that $\mathsf{var}(\tau)$ is included in $\overline{\alpha}$.

Note that we only consider prenex (ML-style) parametric polymorphism, not higher-rank polymorphism, so there are no quantifiers in the syntax of types.

5.1. Subtyping in the polymorphic case: a problem of definition

Before defining formal interpretations for polymorphic types, we briefly review how extending the semantic subtyping framework to the polymorphic case has been addressed in previous work.

The intuition of subtyping in the presence of type variables is that $\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha})$ should hold true whenever, *independently of the variables* $\overline{\alpha}$, any value of type τ_1 has type τ_2 as well. However the correct definition of 'independently' is not obvious. It should look like this:

$$\forall \overline{\alpha}, [\![\tau_1(\overline{\alpha})]\!] \subseteq [\![\tau_2(\overline{\alpha})]\!]$$

but because variables are abstractions, it is not completely clear over what to quantify them. As mentioned in [Hosoya et al. 2009], a candidate — naive — definition would use *ground substitutions*, that is, if the inclusion of interpretations always holds when variables are replaced with ground types, then the subtyping relation holds:

$$\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha}) \Leftrightarrow \forall \overline{\tau} \text{ ground types}, [\![\tau_1(\overline{\tau}/\overline{\alpha})]\!] \subseteq [\![\tau_2(\overline{\tau}/\overline{\alpha})]\!] \quad (2)$$

Obviously the condition on the right must be *necessary* for subtyping to hold. But deciding that it is *sufficient* as well makes the relation unsatisfactory and somehow counterintuitive, as remarked in [Hosoya et al. 2009]. Indeed, suppose int is an *indivisible* type, that is, that it has no subtype beside $\mathbf{0}$ and itself. Then the following would hold:

$$\mathsf{int} \times \alpha \leqslant (\mathsf{int} \times \neg\mathsf{int}) \vee (\alpha \times \mathsf{int}) \tag{3}$$

This relation abuses the definition by taking advantage of the fact that for any ground type τ, either $[\![\mathsf{int}]\!] \subseteq [\![\tau]\!]$ or $[\![\tau]\!] \subseteq [\![\neg\mathsf{int}]\!]$. In the first case, because $[\![\tau]\!] \subseteq ([\![\neg\mathsf{int}]\!] \cup [\![\mathsf{int}]\!])$, we have $[\![\mathsf{int} \times \tau]\!] \subseteq [\![\mathsf{int} \times \neg\mathsf{int}]\!] \cup [\![\mathsf{int} \times \mathsf{int}]\!]$ and then the second member of the union is included in $[\![\tau \times \mathsf{int}]\!]$. In the second case, we directly have $[\![\mathsf{int} \times \tau]\!] \subseteq [\![\mathsf{int} \times \neg\mathsf{int}]\!]$.

This trick, which only works with indivisible ground types, not only shows that candidate definition (2) yields bizarre relations where a variable occurs in unrelated positions on both sides. It also means the candidate definition is very sensitive to the precise semantics of base types, since it distinguishes indivisible types from others. More precisely, it means that refining the collection of base types, for example by adding types even and odd, can break subtyping relations which held true without these new types — this is simply due to the fact that it increases the set over which $\overline{\tau}$ is quantified in (2), making the relation stricter. This could hardly be considered a nice feature of the subtyping relation.

The conclusion is thus that the types in (3) should be considered related *by chance* rather than by necessity, hence not in the subtyping relation, and that quantifying over all possible ground types is not enough; in other words, candidate definition (2) is too weak and does not properly reflect the intuition of 'independently of the variables'. Indeed, (3) is in fact dependent on the variable as we saw, the point being that there are only two cases and that the convoluted right-hand type is crafted so that the relation holds in both of them, though for different reasons.

In order to restrict the definition of subtyping, [Hosoya et al. 2009], which concentrates on XML types, uses a notion of *marking*: some parts of a value can be marked (using paths) as corresponding to a variable, and the relation 'a value has a type' is changed into 'a marked value matches a type', so the semantics of a type is not a set of values but of pairs of a value and a marking. This is designed so that it integrates well in the XDuce language, which has pattern-matching but no higher-order functions (hence no arrow types), so their system is tied to the operational semantics of matching and provides only a partial solution.

The question of finding the correct definition of semantic subtyping in the polymorphic case was finally settled very recently by Castagna and Xu [Castagna and Xu 2011]. Their definition does, in the same way as (2), follow the idea of a universal quantification over possible *meanings* of variables but solves the problem raised by (3) by using a much larger set of possible meanings — thus yielding a stricter relation. More precisely, variables are allowed to represent not just ground types but any arbitrary part of the semantic domain; furthermore, the semantic domain itself must be large enough, which is embodied by the notion of *convexity*. We refer the reader to [Castagna and Xu 2011] for a detailed discussion of this property and its relation to the notion of *parametricity* studied by Reynolds in [Reynolds 1983]; we will here limit ourselves to introducing the definitions strictly necessary for the discussion at hand.

In this work, we do not use this definition with its universal quantification directly. Rather, we retain from [Hosoya et al. 2009] the idea of tagging (pieces of) values which correspond to variables, but do so in a more abstract way, by extending the semantic domain, and define a *fixed* interpretation of polymorphic types in this extended domain as a straightforward extension of the monomorphic framework. We then show how to build a set-theoretic model of polymorphic types, in the sense of [Castagna and Xu 2011], based on this domain, and prove that the inclusion relation on fixed inter-

pretations is equivalent to the full subtyping relation induced by this model. Finally, we explain briefly the notion of convexity and show that this model is convex, implying that this relation is, in fact, the semantic subtyping relation on polymorphic types, as defined in [Castagna and Xu 2011]. These steps are formally detailed in the following section.

5.2. Interpretation of polymorphic types

Let Λ be an infinite set of optional labels, and ι an injective function from the set of variables $\mathcal{V}$ to Λ. (It would be possible to set $\Lambda = \mathcal{V}$, but for clarity we prefer to distinguish *labels* which tag elements of the semantic domain from *variables* which occur in types.) We extend the grammar of (extended) trees by allowing any node to bear, in addition to its single σ label from $\Sigma \cup \{(\to), (\times), \mathbb{B}, \Omega\}$, any (finite) number of labels from Λ. We write it $\sigma_L[tl]$ where L is a finite part of Λ. We extend $\mathcal{C}$ and $\mathcal{D}$ accordingly. When using the non-tree form of types, for instance (d_1, d_2), we indicate the set of root labels on the bottom right like this: $(d_1, d_2)_L$ (here L is the set of labels borne by the $(\times)$ node constituting the root of the pair tree).

We then extend the predicate defining the interpretation of types given in Section 2.2.2 with the following additional case:

$$(\sigma_L[tl] : \alpha) = \iota(\alpha) \in L$$

In other words, the interpretation of a type variable is the set of all trees whose root bears the label corresponding to that variable. The other cases are unchanged, except that the semantic domain is now much larger. This means that the same definition leads to larger interpretations; in particular, the interpretation of a (nonempty) ground type is always an infinite set which contains all possible labellings for each of its trees.

Subtyping over polymorphic types is then defined, as before, as set inclusion between interpretations:

$$\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha}) \Leftrightarrow [\![\tau_1(\overline{\alpha})]\!] \subseteq [\![\tau_2(\overline{\alpha})]\!] \tag{4}$$

It may seem strange to give type *variables* a *fixed* interpretation, and on the other hand it may seem surprising that this definition of subtyping does not actually contain any quantification and is nevertheless stronger than (2) which contains one. The key point is that a form of universal quantification is implicit in the extension of the semantic domain: in some sense, the interpretation of a variable represents all possible values of the variable *at once*. Indeed, for any variable α and any tree d in the domain, there always exist both an infinity of copies of d which are in the interpretation of α and another infinity of copies which are not. From the point of view of logical satisfiability, this makes the domain big enough to contain all possible cases.

In order to show that, despite the appearances, Definition (4) accurately represents a relation that holds *independently of the variables*, we rely, as discussed above, on the formal framework developed by Castagna and Xu [Castagna and Xu 2011]. For this, we first introduce *assignments* η: functions from $\mathcal{V}$ to $\mathcal{P}(\mathcal{D})$ (where $\mathcal{D}$ is the extended semantic domain with labels). Thus an assignment attributes to each variable an arbitrary set of elements from the semantic domain.

We then define the interpretation of a type *relative to an assignment* in the following way: the predicate $(d' :_\eta \tau)$ is defined inductively in the same way as the $(d' : \tau)$ of Section 2.2.2 but with the additional clause:

$$(d :_\eta \alpha) = d \in \eta(\alpha).$$

The interpretation of the polymorphic type τ relative to the assignment η is then $[\![\tau]\!]\eta = \{d \mid (d :_\eta \tau)\}$. This defines an infinity of possible interpretations for a type, depending on the actual values assigned to the variables, and constitutes a set-theoretic model of

types in the sense of [Castagna and Xu 2011]. The subtyping relation induced by this model is the following:

$$\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha}) \Leftrightarrow \forall \eta \in \mathcal{P}(\mathcal{D})^{\mathcal{V}}, [\![\tau_1(\overline{\alpha})]\!]\eta \subseteq [\![\tau_2(\overline{\alpha})]\!]\eta \tag{5}$$

which we can more easily compare to the candidate definition (2): it does in the same way quantify over possible meanings of the variables but uses a much larger set of possible meanings, yielding a stricter relation. We will now prove that this relation is, for our particular model, actually equivalent to (4).

For this, let us first define the *canonical assignment* η_ι as follows:

$$\eta_\iota(\alpha) \stackrel{\text{def}}{=} \{\sigma_L[tl] \in \mathcal{D} \mid \iota(\alpha) \in L\}.$$

Then it is easily seen that the fixed interpretation $[\![\tau]\!]$ of a polymorphic type is the same as its interpretation relative to the canonical assignment, $[\![\tau]\!]\eta_\iota$. What we would like to prove is that the canonical assignment is somehow representative of all possible assignments, making the fixed interpretation sufficient for the purpose of defining subtyping. This is done by the following lemma and corollary.

LEMMA 5.1. *Let V be a finite part of $\mathcal{V}$. Let η be an assignment. Let T be the set of all types τ such that* $\mathsf{var}(\tau) \subseteq V$. *Then there exists a function $F_V^\eta : \mathcal{D} \to \mathcal{D}$ such that: $\forall \tau \in T, \forall d \in \mathcal{D}, d \in [\![\tau]\!]\eta \Leftrightarrow F_V^\eta(d) \in [\![\tau]\!]\eta_\iota$.*

PROOF. For d in $\mathcal{D}$, let $L(d) = \{\iota(\alpha) \mid \alpha \in V \wedge d \in \eta(\alpha)\}$. Since V is finite, $L(d)$ is finite as well. We define $F_V^\eta(d)$ inductively as follows:

— If $d = \mathbb{B}_L[tl]$ then $F_V^\eta(d) = \mathbb{B}_{L(d)}[tl]$
— If $d = (d_1, d_2)_L$ then $F_V^\eta(d) = (F_V^\eta(d_1), F_V^\eta(d_2))_{L(d)}$
— $F_V^\eta(\Omega) = \Omega$
— If $d = \{(d_1, d_1'), \ldots, (d_n, d_n')\}_L$ then
$F_V^\eta(d) = \{(F_V^\eta(d_1), F_V^\eta(d_1')), \ldots, (F_V^\eta(d_n), F_V^\eta(d_n'))\}_{L(d)}$

So F_V^η preserves the structure but changes the labels so that the root node of $F_V^\eta(d)$ is labelled with $L(d)$ and so on inductively for its subterms.

Let $\mathcal{P}(d, \tau) = d \in [\![\tau]\!]\eta \Leftrightarrow F_V^\eta(d) \in [\![\tau]\!]\eta_\iota$. We prove that it holds for all pairs (d, τ) such that τ is in T by induction on those pairs, using the ordering relation $\trianglelefteq$ defined in Section 2.2.2, noticing that $\tau \in T$ implies that all subterms (and unfoldings) of τ are in T as well. The base cases are:

— If τ is a variable. Then it is in V by hypothesis and $\mathcal{P}(d, \tau)$ is true by definition of $L(d)$.
— If it is a base type. Then $\mathcal{P}(d, \tau)$ is true because the interpretation of τ is independent of assignments and labellings.

For the inductive cases, we suppose the property true for all strictly smaller pairs (d, τ) such that τ is in T.

— For the arrow and product cases, the inductive definition of F_V^η makes the result straightforward.
— For the negation and disjunction cases, the result is immediate from the induction hypothesis.
— For $\mu v.\tau$, recall that the well-formedness constraint on types implies that the type's unfolding has a strictly smaller shallow depth than the original type, hence we can use the induction hypothesis on the unfolding and conclude.

□

COROLLARY 5.2. *Let τ be a type.* $\bigcup_{\eta \in \mathcal{P}(\mathcal{D})^{\mathcal{V}}} [\![\tau]\!]\eta = \emptyset$ *if and only if* $[\![\tau]\!]\eta_\iota = \emptyset$.

PROOF. If the union is not empty, there exists η and d such that $d \in [\![\tau]\!]\eta$. From the previous lemma we then have $F^{\eta}_{\mathsf{var}(\tau)}(d) \in [\![\tau]\!]\eta_\iota$. □

This corollary shows that the canonical assignment is representative of all possible assignments and implies that the subtyping relation defined by (4) is equivalent to the one defined by (5).

Convexity of the model.. Definition (5) corresponds to semantic subtyping as defined in [Castagna and Xu 2011], but only on the condition that the underlying model of types be *convex*. Indeed, we can see that this definition is dependent on the set of possible assignments, which itself depends on the chosen (abstract) semantic domain, so it is reasonable to think that increasing the semantic domain could restrict the relation further. In other words, for the definition to be correct, the domain must be large enough to cover all cases. Castagna and Xu's *convexity* characterises this notion of 'large enough'. The property is the following: a set-theoretic model of types is *convex* if, whenever a finite collection of types τ_1 to τ_n each possess a nonempty interpretation relative to some assignment, then there exists a common assignment making all interpretations nonempty at once. This reflects the idea that there are enough elements in the domain to witness all the cases.

In our case, it comes as no surprise that the extended model of types is convex since any nonempty ground type has an infinite interpretation, which, as proved in [Castagna and Xu 2011], is a sufficient condition. But we need not even rely on this result since Corollary 5.2 proves a property even stronger than convexity: having a nonempty interpretation relative to *some* assignment is the same as having a nonempty interpretation relative to *the* common canonical assignment. This stronger property makes the apparently weaker relation defined by (4) equivalent, in our particular model, to the full semantic subtyping relation Castagna and Xu defined. This allows us to reduce the problem of deciding their relation to a question of inclusion between fixed interpretations, making the addition of polymorphism a mostly straightforward extension to the logical encoding we presented for the monomorphic case.

Interestingly, in [Castagna and Xu 2011] the authors suggest that convexity constrains the relation enough that it should allow reasoning on types, similarly to the way parametricity allowed Wadler [Wadler 1989] to deduce 'theorems for free' from typing information. The fact that our logical reasoning approach very naturally has this convexity property — indeed, it is difficult to think of a logical representation of variables which would not have it — seems to corroborate their intuition, although reasoning on types beyond deciding subtyping is currently left as future work.

We now show how the type system extended with type variables is encoded in our logic.

5.3. Logical encoding of variables

We extend the logic with atomic propositions α which behave similarly as σ except they are not mutually exclusive. The interpretation of these propositions is defined as:

$$[\![\alpha]\!] = \{(\sigma_L[tl], c) \mid \iota(\alpha) \in L\}$$

$$[\![\neg\alpha]\!] = \{(\sigma_L[tl], c) \mid \iota(\alpha) \notin L\}$$

The translation $\mathsf{form}(\tau)$ of types into formulas is extended in the obvious way by $\mathsf{form}(\alpha) = \alpha$.

THEOREM 5.3. *With these extended definitions,* $\mathbb{F}[\![\mathsf{fullform}(\tau)]\!] = [\![\tau]\!]$.

PROOF. Preliminary remark: whenever φ does not contain any $\langle 2\rangle$ at toplevel (which is the case of the formulas representing types), then $[\![\varphi]\!] = \mathbb{F}[\![\varphi]\!] \times \mathbf{C}$ where $\mathbf{C}$ is the set of all possible contexts. Hence, when considering such formulas, set-theoretic relations between full interpretations are equivalent to the same relations between first components.

First we check that $\mathbb{F}[\![\mathsf{isd}]\!] = \mathcal{D}$ and reformulate the statement as $\mathcal{D} \cap \mathbb{F}[\![\mathsf{form}(\tau)]\!] = [\![\tau]\!]$.

We make the embedding function tree explicit for greater clarity. What we have to show is that, for any d in $\mathcal{D}$, we have $(d : \tau)$ if and only if $(\mathsf{tree}(d), c)$ is in $[\![\mathsf{form}(\tau)]\!]$ for some (or, equivalently, for any) c.

The property is proved by induction on the pair (d, τ), following the definition of the predicate:

— For $(c : b)$ it holds by definition.
— For $((d_1, d_2)_L : \tau_1 \times \tau_2)$, let $f = (\mathsf{tree}((d_1, d_2)_L), c)$. f is in $[\![\mathsf{form}(\tau_1 \times \tau_2)]\!]$ if and only if $f\langle 1\rangle$ is in $[\![\mathsf{form}(\tau_1)]\!]$ and $f\langle 1\rangle\langle 2\rangle$ is in $[\![\mathsf{form}(\tau_2)]\!]$. (We already know that the node name is $(\times)$ by the structure of d.) Just see that the tree rooted at $f\langle 1\rangle$ is $\mathsf{tree}(d_1)$ and the one at $f\langle 1\rangle\langle 2\rangle$ is $\mathsf{tree}(d_2)$.
— For functions, use the finite unfolding property and the fact the set of pairs is finite, then see, similarly as above, that the correct properties are enforced when navigating the tree.
— For union, negation and empty types, use the preliminary remark.
— For $(d : \alpha)$, just see that $d \in \iota(\alpha)$ and $d \in \mathbb{F}[\![\alpha]\!]$ both mean that the root node of d, which is the node at focus in the formula, bears the label $\iota(\alpha)$.
— For $(d : \mu v.\tau)$, use the property that the interpretation of a fixpoint formula and its unfolding are the same (lemma 4.2 of [Genevès et al. 2007]).

□

COROLLARY 5.4. *$\tau_1 \leqslant \tau_2$ holds if and only if* $\mathsf{fullform}(\tau_1 \wedge \neg\tau_2)$*, or alternatively* $\mathsf{isd} \wedge \mathsf{form}(\tau_1) \wedge \neg\mathsf{form}(\tau_2)$*, is unsatisfiable.*

5.4. Complexity

LEMMA 5.5. *Provided two types τ_1 and τ_2, the subtyping relation $\tau_1 \leqslant \tau_2$ can be decided in time $2^{\mathcal{O}(|\tau_1|+|\tau_2|)}$ where $|\tau_i|$ is the size of τ_i.*

PROOF. The logical translation of types performed by the function $\mathsf{form}(\cdot)$ does not involve duplication of subformulas of variable size, therefore $\mathsf{form}(\tau)$ is of linear size with respect to $|\tau|$. Since isd has constant size, the whole translation $\mathsf{fullform}(\tau)$ is linear in terms of $|\tau|$. For testing satisfiability of the logical formula, we use the satisfiability-checking algorithm presented in [Genevès et al. 2007] whose time complexity is $2^{\mathcal{O}(n)}$ in terms of the formula size n. □

6. IMPLEMENTATION AND PRACTICAL EXPERIMENTS

In this section we report on some interesting lessons learned from practical experiments with the implementation of the system in order to prove relations in the type algebra. We first describe the main techniques used to implement the whole system, the minimal necessary background for using the implementation, and then we review and discuss several informative examples.

6.1. Implementation Principles

The algorithm for deciding the subtyping relation has been implemented on top of the satisfiability solver that was first introduced in [Genevès 2006; Genevès et al. 2007].

Since this algorithm constitutes the core of our implementation we briefly review its essential principles below and highlight its properties in the polymorphic setting.

Search universe and exponential complexity.. The fundamental principle of the algorithm is to look for a finite tree that satisfies a given logical formula. For this purpose, it first constructs a compact representation of the relevant search universe in which to look for a tree model satisfying the given formula. This representation, called the Lean of the formula, is a set of subformulas of the initial formula. It is computed from the so-called Fisher-Ladner closure of the initial formula: it is composed of all the atomic propositions found in the formula, plus all distinct modal subformulas that can be obtained by unrolling fixpoints, and four basic "topological" formulas that indicate whether a given node admits some parent node, some child (or whether it is leaf and/or a root). This Lean set is important since its powerset precisely defines the search universe in which the algorithm looks for trees. For this reason, the time complexity of the algorithm is $2^{\mathcal{O}(n)}$ with respect to Lean size n. The acute reader may notice that the Lean size of a large logical formula is usually smaller than the size of the formula measured as the number of all connectives and operands. This is because the Lean representation naturally eliminates duplicate subformulas and discards disjunctions and conjunctions at top level. Furthermore, we implemented an additional optimization: we rely on a binary encoding of indices of symbols in order to perform a logarithmic compression of the number of atomic propositions in the lean. The effect of this optimization can be observed for complex types with a large number of symbols, for which it is particularly effective.

Bottom-up search as a fixpoint computation.. Once the Lean set is known, the algorithm starts traversing all relevant tree nodes in an attempt to build a satisfying tree. This search is performed in a bottom-up fashion, in the manner of a fixpoint computation. The algorithm considers a set of tree nodes whose subtrees have been proved consistent. The algorithm begins with the empty set of nodes, then at the first step, all possible leaves are added. Then, the algorithm repeatedly try to add new tree nodes to this set, until no more nodes can be added, i.e. a fixpoint has been reached. It is easy to observe that the algorithm terminates since, in the worst case (when the formula is unsatisfiable) it explores all the relevant nodes, that is, all subsets of the Lean, which is a finite set. At each step, whenever the algorithm is about to add a candidate node to the set of proved nodes, essential checks are performed to make sure that the higher tree rooted at the candidate node is logically consistent with subtrees already proved at earlier steps. In particular, modal formulas may impose constraints on successor nodes that must be checked for consistency when two nodes are connected. These checks are described formally in [Genevès et al. 2007]. At each step of the computation, the truth status of the initial formula given as input to the algorithm is tested at the freshly proved nodes. It the formula is found to hold at this node then the algorithm immediately terminates with a proof that the formula is satisfiable. This step by step approach offers several advantages. First, it opens the door to an implementation with semi-implicit techniques, and second, one can easily keep track of the current state of the set of proved nodes at each step in order to generate small satisfying trees.

Use of semi-implicit techniques.. An important observation about the fixpoint computation is that for a given candidate node to be added to the set of proved nodes, the algorithm does not need to keep track of all possible subtrees that are consistent with the candidate node, but instead it is enough to find only one proved subtree for each

This optimization can be turned on using the `-compressElts` argument on the command line offline version of the solver.

successor of the candidate node. This observation has an important consequence: it makes it possible to avoid the explicit enumeration of all proved subtrees into memory. Instead checking the existence of at least one proved subtree per required successor of a candidate node is enough. This makes it possible to encode the algorithm with boolean functions operating on a bit-vector representation of the Lean set (as described in [Genevès 2006]), opening the door for an implementation based on Binary Decision Diagrams (BDDs) [Bryant 1986]. BDDs provide a canonical representation of boolean functions. Experience has shown that this representation is very compact for very large Boolean functions. Furthermore, the effectiveness of operations over BDDs is notably well-known in the area of formal verification of systems [Edmund M. Clarke et al. 1999], in the context of simpler (less expressive) modal logics like $\mathcal{K}$ [Pan et al. 2006], and even in the context of much more complex problems that can be reduced to μ-calculus satisfiability testing, such as the problem of automatically detecting the impacts of a schema change on a regular query [Genevès et al. 2009]. Here again, the use of BDDs constitutes one of the major reasons why our approach performs well in practice.

Generation of Counter-Examples.. The role of the satisfiability-solving algorithm is not limited to the partitioning of the set of logical formulas based on whether they are satisfiable or not: it can in addition generate a sample satisfying tree for satisfiable formulas. Technically, once the formula is found satisfiable at some node, the implementation reconstructs a sample satisfying tree in a top-down manner, starting from the root of the satisfying tree. It actually attempts to generate one of the smallest possible satisfying trees. For that purpose, a pointer to the current state of the set of proved nodes is kept at each step of the fixpoint computation. During the (re)construction of the satisfying tree, smaller proved subtrees are then preferred, resulting in a minimal satisfying tree.

In the context of our type algebra, the validity of a subtyping statement of the form $\tau_1 \leqslant \tau_2$ is checked by testing for the unsatisfiability of $\psi = \neg(\tau_1 \leqslant \tau_2)$. If ψ is unsatisfiable then τ_1 is a subtype of τ_2. If ψ is satisfiable, then the tree satisfying ψ generated by the algorithm represents a counter-example for the relation $\tau_1 \leqslant \tau_2$. Such a sample tree often happens to be of great practical value in order to ease the understanding of the reason(s) why the relation does not hold.

In the polymorphic setting, a counter-example is in principle, according to the extended semantics, a labelled tree. However, as mentioned in Section 5.2, whenever a formula is satisfiable there always exists an infinity of possible labellings which satisfy it. Therefore, rather than proposing just one labelled tree, the solver gives a minimal tree together with *labelling constraints* representing all labellings which make that particular tree a counter-example. Namely, for each variable α, every node will be labelled with α to indicate that it *must* be labelled with α for the formula to be satisfied, with $\neg\alpha$ to indicate that it *must not* be, or with nothing if label α is irrelevant for that particular node. This allows an easier interpretation of the counter-example in terms of assignments: the subtyping relation fails whenever the assignment for each variable α contains all the trees whose root is marked with α and none of those whose root is marked with $\neg\alpha$.

6.2. Using the Implementation

Our implementation is publicly available. Interaction with the system is offered through a user interface in a web browser. The whole system is available online at:

http://wam.inrialpes.fr/websolver/

A screenshot of the interface is given in Figure 2. The user can either enter a formula through area (1) of Figure 2 or select from pre-loaded analysis tasks offered in area (4)

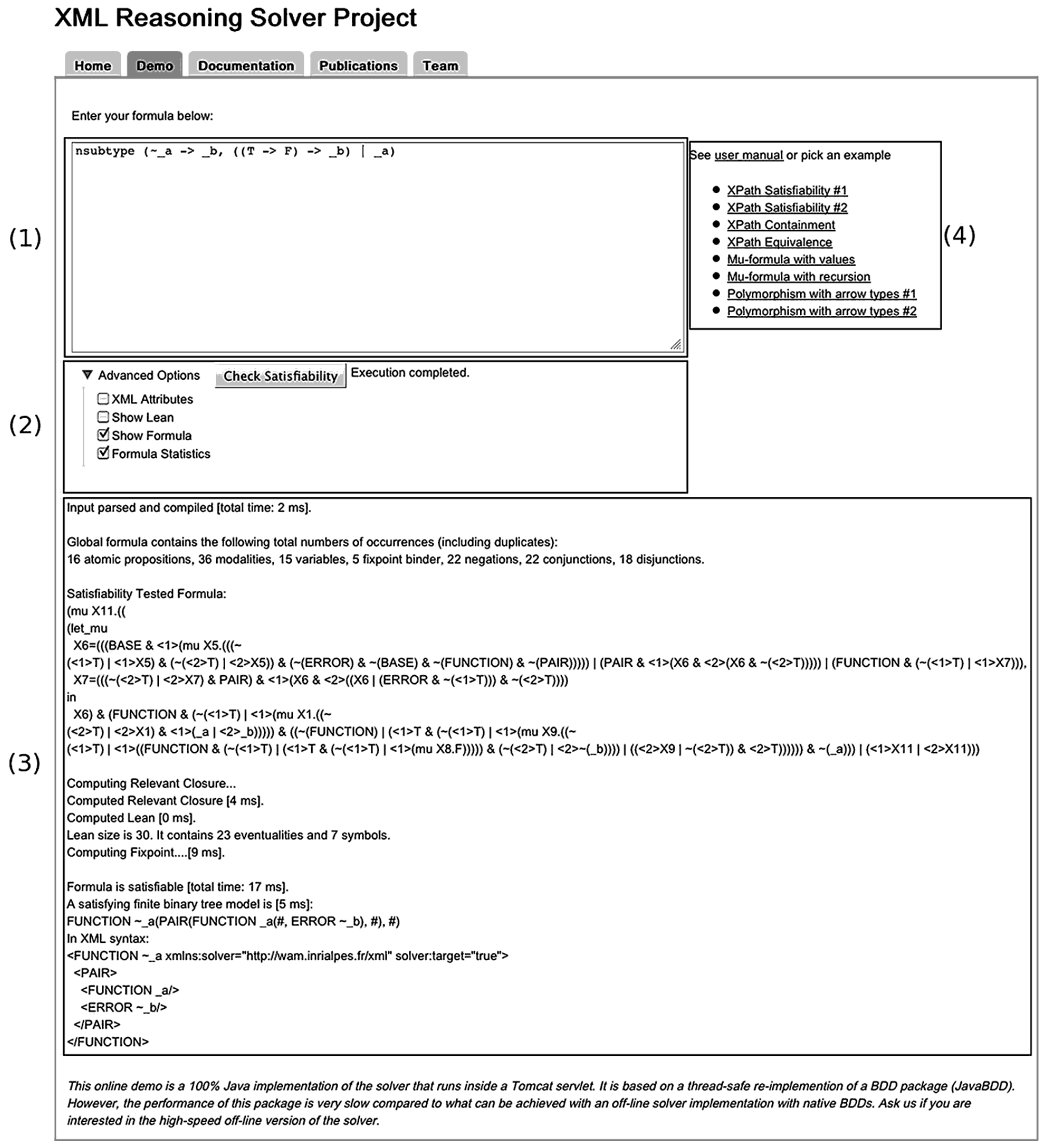

Fig. 2. Screenshot of the Interface.

of Figure 2. The level of details displayed by the solver can be adjusted in area (2) of Figure 2 and makes it possible to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 2 together with counter-examples.

Concrete Syntax for Type Algebra.. All the examples in the subsection that follows can be tested in our online prototype. For this purpose, the following table gives the correspondence between the syntax used in the paper and the syntax that must be used in the implementation:

```
(mu X8.(((
(let_mu
  X5=(((BASE & <1>(mu X4.(((~(<1>T) | <1>X4) & (~(<2>T) | <2>X4))
     & (~(ERROR) & ~(BASE) & ~(FUNCTION) & ~(PAIR)))))
       | (PAIR & <1>(X5 & <2>(X5 & ~(<2>T))))) | (FUNCTION & (~(<1>T) | <1>X6))),
  X6=(((~(<2>T) | <2>X6) & PAIR) & <1>(X5 & <2>((X5 | (ERROR & ~(<1>T))) & ~(<2>T))))
in
  X5) & ((FUNCTION & (~(<1>T) | <1>(mu X1.((~(<2>T) | <2>X1) & <1>(~(_a) | <2>_g)))))
      & (FUNCTION & (~(<1>T) | <1>(mu X2.((~(<2>T) | <2>X2) & <1>(~(_b) | <2>_g)))))))
       & (~(FUNCTION) | (<1>T & (~(<1>T) | <1>(mu X7.((<2>T & (~(<2>T) | <2>X7))
        | (~(<1>T) | <1>((_a | _b) & (~(<2>T) | <2>~(_g))))))))) | (<1>X8 | <2>X8)))
```

Fig. 3. Logical translation tested for satisfiability.

	Paper Syntax	Implementation Syntax
Type variables	α, β, γ	`_a, _b, _g`
Type constructors	$\times, \rightarrow$	`*, ->`
Recursive types	$\mu v.\tau$	`let $v = t in $v`
Basic types	$\mathbf{0}, \mathbf{1}$	`F, T`
Logical connectives	$\wedge, \vee, \neg, \Rightarrow$	`&, |, ~, =>`
Subtyping	$\neg(\tau_1 \leqslant \tau_2)$	`nsubtype(t1,t2)`

Additionally, the embedding of a base formula of the logic into a base type is provided by curly braces: $\{\varphi\}$ is an abbreviation for $\mathsf{isbase} \wedge \langle 1 \rangle \varphi$.

6.3. Examples and Discussion

The goal of this subsection is to illustrate through some examples how our logical setting is natural and intuitive for proving subtyping relations. For example, one can prove simple properties such as the one below:

$$(\alpha \rightarrow \gamma) \wedge (\beta \rightarrow \gamma) \leqslant (\alpha \vee \beta) \rightarrow \gamma \tag{6}$$

This is formulated as follows:

```
nsubtype((_a -> _g) & (_b -> _g), (_a | _b) -> _g)
```

which is automatically compiled into the logical formula shown on Figure 3 and given to the satisfiability solver that returns:

```
Formula is unsatisfiable [16 ms].
```

which means that no satisfying tree was found for the formula, or, in other terms, that the negation of the formula is valid. The satisfiability solver is seen as a theorem prover since its run built a formal proof that property (6) holds.

Lists.. Jerôme Vouillon [Vouillon 2006] uses simple examples with lists to illustrate polymorphism with recursive types. For instance, consider the type of lists of elements of type α:

$$\tau_{\text{list}} = \mu v.(\alpha \times v) \vee \text{nil}$$

where "nil" is a singleton type. The type of lists of an even number of such elements can be written as:

$$\tau_{\text{even}} = \mu v.(\alpha \times (\alpha \times v)) \vee \text{nil}$$

By giving the following formula to the solver :

```
nsubtype(let $v = (_a * _a * $v) | {nil} in $v,
               let $w= (_a * $w) | {nil} in $w )
```

which is found unsatisfiable, we prove that

$$\tau_{\text{even}} \leqslant \tau_{\text{list}}$$

If we now consider the type of lists of an odd number of elements of type α:

$$\tau_{\text{odd}} = \mu v.(\alpha \times (\alpha \times v)) \vee (\alpha \times \text{nil})$$

we can check additional properties in a similar manner, like:

$$(\tau_{\text{even}} \vee \tau_{\text{odd}} \leqslant \tau_{\text{list}}) \wedge (\tau_{\text{list}} \leqslant \tau_{\text{even}} \vee \tau_{\text{odd}})$$

The following formula corresponds to the example (1) of the introduction:

```
bool() = {true|false};
list() = let $l = (_a * $l) | {nil} in $l;
odd()  = let $o = (_a * _a * $o) | (_a * {nil}) in $o;
even() = let $e = (_a * _a * $e) | {nil} in $e;

nsubtype ( (odd() -> {true}) & (even() -> {false}),
           list() -> bool() )
```

This formula is found unsatisfiable by the solver, which proves the validity of the subtyping statement (1).

Hints about non-trivial relations.. Giuseppe Castagna (see section 2.7 of [Castagna and Xu 2011]) gives some examples of non-trivial relations that hold in the type algebra. For instance, the reader can check that the types $\mathbf{1} \to \mathbf{0}$ and $\mathbf{0} \to \mathbf{1}$ can be seen as extrema among the function types:

$$\mathbf{1} \to \mathbf{0} \leqslant \alpha \to \beta \quad \text{and} \quad \alpha \to \beta \leqslant \mathbf{0} \to \mathbf{1}$$

Our system also permitted to detect an error in [Castagna and Xu 2011] and provided some helpful information to the authors of [Castagna and Xu 2011] in order to find the origin of the error and make corrections. Specifically, in a former version of [Castagna and Xu 2011], the following relation was considered:

$$(\neg\alpha \to \beta) \leqslant ((\mathbf{1} \to \mathbf{0}) \to \beta) \vee \alpha \tag{7}$$

Authors explained how this relation was proved by their algorithm. However, by encoding the relation in our system we found that this relation actually does not hold. Specifically, this is formulated as follows in our system:

```
nsubtype (~_a -> _b, ((T -> F) -> _b) | _a)
```

The satisfiability solver, when fed this formula, returns the following counter-example:

```
FUNCTION ~_a (PAIR(FUNCTION _a (#,  ~_b ERROR), #), #)
```

`FUNCTION` represents $(\to)$ and `PAIR` represents $(\times)$. This is a binary tree representation of the n-ary tree

$$(\to)_{\neg\alpha}[(\times)[(\to)_{\alpha}[\epsilon] :: \Omega :: \epsilon] :: \epsilon]$$

which corresponds to the domain element

$$\{(\{\}_{\alpha}, \Omega)\}_{\neg\alpha}.$$

The inner $(\to)$ node has no children and thus represents the function which always diverges: $\{\}$. More precisely, it represents *a* copy f of this function that belongs to the interpretation of α. The root $(\to)$ node then represents a function which is *not* in $[\![\alpha]\!]$ and which to f associates an error, while diverging on any other input.

Now, why is it a counter-example to (7)? As the function diverges but on one input f and that input is in $[\![\alpha]\!]$, it is vacuously true that on all inputs in $[\![\neg\alpha]\!]$ for which it

returns a result, this result is in $[\![\beta]\!]$. Thus it does have the type on the left-hand side. However, it does not have type α, nor does it have type $((\mathbf{1} \to \mathbf{0}) \to \beta)$. Indeed, f does have type $\mathbf{1} \to \mathbf{0}$ and our counter-example function associates to it an error, which is not in $[\![\beta]\!]$.

Big XML Types.. One purpose of the next example is to illustrate how the system can handle large types. We consider the tree grammars that define the admissible structures of webpages conforming to the XHTML Basic 1.0 and XHTML Basic 1.1 specifications. We formulate a very simple example to check that the type of a function that modifies XHTML Basic 1.0 documents while preserving their validity is a subtype of the type of a more general transformation that produces XHTML Basic 1.1 output from XHTML Basic 1.0 input documents.

The formula given to the solver is the following:

```
nsubtype({type("xhtml-basic10.dtd","html")}->{type("xhtml-basic10.dtd","html")},
         {type("xhtml-basic10.dtd","html")}->{type("xhtml-basic11.dtd", "html")})
```

The system first triggers the parsing of the two real world tree grammars whose sizes are significant: xhtml-basic10.dtd contains 52 tag names and 71 type variables, whereas xhtml-basic11.dtd contains 67 tag names and 89 type variables. Those grammars are then linearly compiled into logical formulas in order to compose the global logical formula. The entire expansion of this global formula is huge: it contains 1461 atomic propositions, 7150 modalities, 2410 variables, 8 (n-ary) fixpoint binders, 3715 negations, 2126 conjunctions and 3226 disjunctions. Those numbers include duplicate subformulas that are factorized by our lean set representation, as described in Section 6.1. The intermediate results are as follow:

```
Input parsed and compiled [total time: 1562 ms].
Computing Relevant Closure...
Computed Relevant Closure [533 ms].
Computed Lean [2 ms].
Lean size is 201. It contains 192 modal formulas and 9 symbols.
```

This means that the global search universe in which a tree is automatically looked for is composed of 2^{201} distinct tree nodes! Nevertheless, the fixpoint is computed in less than 18 seconds:

```
Fixpoint Computation Initialized [149 ms].
Computing Fixpoint...........[16542 ms].
Formula is unsatisfiable [17229 ms].
```

7. RELATED WORK

We review below related works while recalling how the introduction of XML progressively renewed the interests in parametric polymorphism.

The seminal work by Hosoya, Vouillon and Pierce on a type system for XML [Hosoya et al. 2005] applied the theory of regular expression types and finite tree automata in the context of XML. The resulting language XDuce [Hosoya and Pierce 2003] is a strongly typed language featuring recursive, product, intersection, union, and complement types. The subtyping relation is decided through a reduction to containment of

---

The command line instruction to reproduce this test with the offline version of the solver is the following: `java -Xmx5g -Xms2g -Xmx2g -jar solver.jar fleche.txt -compressElts -stats` where the first three arguments increase the default heap size of the java virtual machine, "`fleche.txt`" is a text file containing the logical formula, "`-compressElts`" indicates that the binary encoding of symbols must be used in order to reduce the lean size, and "`-stats`" displays some formula statistics like e.g. the number of different connectives in the logical formula.

finite tree automata, which is known to be in EXPTIME. This work does not support function types nor polymorphism, but provided a ground for further research.

In particular, Frisch, Castagna and Benzaken provide a gentle introduction to semantic subtyping in [Frisch et al. 2008]. Semantic subtyping focuses on a set-theoretic interpretation, as opposed to traditional subtyping through direct syntactic rules. Our logical modeling presented in Section 4 naturally follows the semantic subtyping approach as the underlying logic has a set-theoretic semantics. Frisch, Castagna and Benzaken added function types to the semantic subtyping performed by XDuce's type system. This notably resulted in the CDuce language [Benzaken et al. 2003]. However, CDuce does not support type variables and thus lacks polymorphism.

Vouillon studied polymorphism in the context of regular types with arrow types in [Vouillon 2006]. Specifically, he introduced a pattern algebra and a subtyping relation defined by a set of syntactic inference rules. A semantic interpretation of subtyping is given by ground substitution of variables in patterns. The type algebra has the union connective but lacks negation and intersection. The resulting type system is thus less general than ours.

Polymorphism was also the focus of the later work found in [Hosoya et al. 2009]. In [Castagna and Xu 2011], it is explained that at that time a semantically defined polymorphic subtyping looked out of reach, even in the restrictive setting of [Hosoya and Pierce 2003], which did not account for higher-order functions. This is why [Hosoya et al. 2009] fell back on a somewhat syntactic approach linked to pattern-matching that seemed difficult to extend to higher-order functions. Our work shows however that such an extension was possible using similar basic ideas, only slightly more abstract.

The most closely related work is the one found in [Castagna and Xu 2011], in the same proceedings as the current paper, which solves the problem of defining subtyping semantically in the polymorphic case for the first time, and addresses the problem of its decision through an ad-hoc and multi-step algorithm, which was only recently proved to terminate in all cases. Our approach also addresses the problem of deciding their subtyping relation and solves it through a more direct, generic, natural and extensible approach since our solution relies on a modeling into a well-known modal logic (the μ-calculus) and on using a satisfiability solver such as the one proposed in [Genevès et al. 2007]. This logical connection also opens the way for extending polymorphic types with several features found in modal logics.

The work of [Bierman et al. 2010] follows the same spirit than ours: typechecking is subcontracted to an external logical solver. An SMT-solver is used to extend a type-checker for the language Dminor (a core dialect for M) with refinement type and type-tests. The type-checking relies on a semantic subtyping interpretation but neither function types nor polymorphism are considered. Therefore, their work is incomparable to ours.

The present work heavily relies on the work presented in [Genevès et al. 2007] since we repurpose the satisfiability-checking algorithm of [Genevès et al. 2007] for deciding the subtyping relation. The goal pursued in [Genevès et al. 2007] was very different in spirit: the goal was to decide containment of XPath queries in the presence of regular tree types. To this end, the decidability of a logic with converse for finite ordered trees is proved in a time complexity which is a simple exponential of the size of the formula. The present work builds on these results for solving semantic subtyping in the polymorphic case.

8. CONCLUSION

The main contribution of this paper is to define a logical encoding of the subtyping relation defined in [Castagna and Xu 2011], yielding a decision algorithm for it. We prove that this relation is decidable with an upper-bound time complexity of $2^{(n)}$, where

n is the size of types being checked. In addition, we provide an effective implementation of the decision procedure that works well in practice.

This work illustrates a tight integration between a functional language type-checker and a logical solver. The type-checker uses the logical solver for deciding subtyping, which in turn provides counter-examples (whenever subtyping does not hold) to the type-checker. These counterexamples are valuable for programmers as they represent evidence that the relation does not hold. As a result, our solver represents a very attractive back-end for functional programming languages type-checkers.

This result pushes the integration between programming languages and logical solvers to an advanced level. The proposed logical approach is not only capable of modeling higher order functions, but it is also capable of expressing values from semantic domains that correspond to monadic second-order logics such as XML tree types. This shows that such logical solvers can become the core of XML-centric functional languages type-checkers such as those used in CDuce or XDuce.

Acknowledgments.. We are thankful to Giuseppe Castagna for bringing our attention to the problem of subtyping with arrow types in the polymorphic case. He gave us precious insights for the precise formulation of the problem, that he also addressed in a paper published in the current proceedings. Several people also discussed or exchanged with us about subtyping and polymorphism: Zhiwu Xu, Véronique Benzaken and Kim Nguyen.

REFERENCES

Michael Benedikt and James Cheney. 2010. Destabilizers and Independence of XML Updates. *Proceedings of the VLDB Endowment* 3, 1 (2010), 906–917.

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th international conference on functional programming (ICFP '03)*. Uppsala, Sweden, 51–63. http://doi.acm.org/10.1145/944705.944711

Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. 2010. Semantic subtyping with an SMT solver. In *Proceedings of the 15th international conference on functional programming (ICFP '10)*. Baltimore, MD, USA, 105–116. http://doi.acm.org/10.1145/1863543.1863560

Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. 2007. XQuery 1.0: An XML Query Language, W3C Recommendation. (January 2007).

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers* 35, 8 (1986), 677–691.

G. Castagna and Z. Xu. 2011. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *Proceedings of the 16th international conference on functional programming (ICFP '11)*. Tokyo.

James Clark and Steve DeRose. 1999. XML Path Language (XPath) Version 1.0, W3C Recommendation. (November 1999). http://www.w3.org/TR/1999/REC-xpath-19991116

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. 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)*. Budapest, 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24

Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model checking*. MIT Press, Cambridge, MA, USA.

Daniel Engovatov and Jonathan Robie. 2010. XQuery 3.0 Requirements, W3C Working Draft. (September 2010). http://www.w3.org/TR/xquery-30-requirements/

A. Frisch, G. Castagna, and V. Benzaken. 2008. Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 4 (2008), 1–64.

Pierre Genevès. 2006. *Logics for XML*. Ph.D. Dissertation. Institut National Polytechnique de Grenoble. http://wam.inrialpes.fr/publications/2006/geneves-phd.pdf.

Pierre Genevès, Nabil Layaïda, and Vincent Quint. 2009. Identifying query incompatibilities with evolving XML schemas. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 221–230. DOI:http://dx.doi.org/10.1145/1596550.1596583

Pierre Genevès, Nabil Layaïda, and Alan Schmitt. 2007. Efficient Static Analysis of XML Paths and Types. In *Proceedings of the 28th conference on programming language design and implementation (PLDI '07)*. San Diego, CA, USA, 342–351. http://doi.acm.org/10.1145/1250734.1250773

H. Hosoya, A. Frisch, and G. Castagna. 2009. Parametric Polymorphism for XML. *ACM Transactions on Programming Languages and Systems* 32, 1 (2009), 1–56.

Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology* 3, 2 (2003), 117–148. http://doi.acm.org/10.1145/767193.767195

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems* 27 (January 2005), 46–90. Issue 1. http://doi.acm.org/10.1145/1053468.1053470

Guoqiang Pan, Ulrike Sattler, and Moshe Y. Vardi. 2006. BDD-Based Decision Procedures for the modal logic K. *Journal of Applied Non-classical Logics* 16, 1-2 (2006), 169–208.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Jérôme Vouillon. 2006. Polymorphic regular tree types and patterns. In *Proceedings of the 33rd symposium on principles of programming languages (POPL '06)*. Charleston, SC, USA, 103–114. http://doi.acm.org/10.1145/1111037.1111047

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the 4th international conference on functional programming languages and computer architecture (FPCA '89)*. London, 347–359. http://doi.acm.org/10.1145/99370.99404