HAL
open science

A Logical Approach To Deciding Semantic Subtyping

Nils Gesbert, Pierre Genevès, Nabil Layaïda

To cite this version:

Nils Gesbert, Pierre Genevès, Nabil Layaïda. A Logical Approach To Deciding Semantic Subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 2015, 38 (1), pp.31. 10.1145/2812805 . hal-00848023v2

HAL Id: hal-00848023

https://inria.hal.science/hal-00848023v2

Submitted on 17 Aug 2015

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.

A Logical Approach To Deciding Semantic Subtyping

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. We consider the subtyping relation defined by Castagna and Xu [2011] 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. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 00 (0000), 32 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 supply a value of type τ instead [Liskov and Wing 1994]. This property has a natural set-theoretic interpretation: the set of values of type τ is included in the set of values 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, 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, as well as intersection and negation types, in the language CDuce [Benzaken et al. 2003]. In CDuce, 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

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

out to be non-trivial and the recipe for managing it is explained by Frisch et al. [2008] — we summarise it, with a slightly different focus than the original paper, in Section 2.

More recently, Hosoya et al. [2009] extended the XDuce type algebra with type variables so as to support prenex parametric polymorphism. 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 [2011].

In the works of Frisch et al. [2008] and of 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 intersections of arrow types are allowed, but it can still be done. In general, very schematically, the algorithms rely on a coinductive definition of the relation, where subtyping between complex types is related with subtyping between their components. For recursive types, subtyping holds if, no matter how far the types are traversed, no contradiction is ever reached. It thus involves traversing constructors and distinguishing cases repeatedly. 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 seminal 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 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.

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], 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. Polymorphism and Subtyping: an 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 functional language designed for querying collections of XML data. The support of higher-order functions appears in the requirements for the XQuery 3.0 language [Robie et al. 2014]. 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).

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 lists whose elements are of type α. These lists are classically represented by nested pairs, with the empty list being represented by a special constant `nil`. We assume this constant has its own type $\{\texttt{nil}\}$ representing only itself (a 'singleton type'). The type $[\alpha]$ can then be defined recursively in the following way:

$$[\alpha] = \mu v.\{\texttt{nil}\} \vee (\alpha \times v)$$

where $\times$ denotes the cartesian product and μ binds the variable v for denoting a recursive type.

The type τ of functions that process an α-list and return a boolean is written as follows:

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

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:

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

where $\{\texttt{true}\}$ and $\{\texttt{false}\}$ are singleton types. If we make explicit the parametric types even$[\alpha]$ and odd$[\alpha]$, τ' becomes:

$$\tau' = \left(\begin{array}{lll} & \mu v.(\alpha \times (\alpha \times v)) \vee \{\texttt{nil}\} & \to \{\texttt{true}\} \\ \wedge & \mu v.(\alpha \times (\alpha \times v)) \vee (\alpha \times \{\texttt{nil}\}) & \to \{\texttt{false}\} \end{array} \right)$$

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 captured by:

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

where $\leqslant$ denotes the subtyping relation which is under scrutiny in this article. We give more examples in Section 6.

1.2. Static Typing and 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. 2015] are used as basic building blocks for type systems for XPath [Clark and DeRose 1999].

The main idea in this paper is to use a logical satisfiability solver for deciding subtyping. 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 expressions [Genevès et al. 2007].

This fruitful connection between logics, their decision procedures, and programming languages permits 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 the one of 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 solve containment and equivalence of XPath expressions [Genevès et al. 2007];
— Monadic second-order logic solvers over trees, and – equivalent yet much more effective – satisfiability-solvers for μ-calculus over trees [Genevès et al. 2015] are used to solve query containment problems in the presence of type constraints.

1.3. Contributions

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 [2011], and gives a precise complexity upper-bound: $2^{\mathcal{O}(n)}$, 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].

A preliminary version of this work was presented at the International Conference on Functional Programming in 2011 [Gesbert et al. 2011]. This article extends and improves that previous publication in the following ways:

— The type algebra we consider is more expressive since all types definable in the system of Castagna and Xu [2011] are now supported, whereas in our previous paper there were some restrictions on recursive types. In particular, our decision procedure now decides subtyping between types such as $\mu v.(v \to \alpha) \wedge \beta$ and $\mu v.(v \to \alpha) \vee \beta$. More examples of such types as well as subtyping relations between such types are given in Section 6.3.

— Our results rely on an encoding of abstract values as trees. In the previous encoding, those trees were unranked n-ary trees, and this required introducing a form of zipper [Huet 1997] to properly define the semantics of formulas. The encoding we present here uses binary trees instead, which removes the need for zippers and therefore simplifies the whole presentation.

— The semantic subtyping framework of Frisch et al. [2008] is parameterized by a set of basic constants, a set of basic types and the interpretation of the latter in terms of the former. Our previous translation of types into formulas assumed that it was possible to represent basic constants as trees and basic types as formulas (which describe sets of trees) but did not suggest a particular way of doing it. In this paper, we present a different approach where it is unnecessary to encode basic constants, where basic types are represented by abstract symbols, and where the single parameter of the translation is a Boolean formula called basic_constraint describing the relations between basic types. In this way, the parameter is explicit and the relation with the original type language is more apparent.

— The previous encoding of arrow types contained a subtle error which is now corrected (see footnote 3 on page 15 for details).

— The presentation is more detailed overall, and the article is more self-contained.

1.4. Outline

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 by 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

Types are defined starting from a finite set $\mathcal{B}$ of *basic types*, ranged over by b. Typically, these basic types would include things like `integer`, `character`, and some *singleton types* like `true` or `false`, representing abstract constants used to build enumerated 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 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 $\mu v.\neg v$.

Types include, for example, the type of booleans, $\texttt{true} \vee \texttt{false}$, or the type of integer lists, $\mu v.(\texttt{nil} \vee (\texttt{integer} \times v))$ (where we assume $\texttt{nil}$ is a singleton type).

Additionally, the following abbreviations are defined:

$$\tau_1 \wedge \tau_2 = \neg(\neg\tau_1 \vee \neg\tau_2)$$

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 it 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 $\mathcal{W} \times (\mathcal{W} \cup \{\Omega\})$, where Ω is not a value of the language but represents an error. 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 denotation 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. Frisch et al. [2008] call this the *extensional interpretation* of function types. Formally:

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

where $(d : \tau)$ means the value d has type τ. Boolean combinators can be interpreted as the corresponding set-theoretic operations on extensional interpretations[1], 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 we can find a language such that, whenever there exists an abstract function of some type, there is also a function of that type in the language. However the following property makes it easy to define such a language: 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 [Frisch et al. 2008, Lemma 6.32]. 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 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 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[2] 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.

[1]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 of 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 is taken into account.

[2]Using lists of pairs rather than sets of pairs allows a much simpler inductive definition of abstract values and changes nothing to the theory. It simply means that several different values represent functions with exactly the same behaviour.

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 $\mathcal{C}$:

$$
\begin{array}{rlll}
d & ::= & & \text{domain element} \\
 & & c & \text{base constant} \\
 & | & (d, d) & \text{pair} \\
 & | & \mathrm{f} & \text{function} \\
\mathrm{f} & ::= & & \text{function} \\
 & & [] & \text{diverging function (empty list)} \\
 & | & ((d, d') :: \mathrm{f}) & \text{finite function (list of pairs)} \\
d' & ::= & & \text{function result} \\
 & & d & \text{domain element} \\
 & | & \Omega & \text{error}
\end{array}
$$

We suppose we have an interpretation $\mathbb{B}[\![\cdot]\!] : \mathcal{B} \to \mathcal{P}(\mathcal{C})$ of basic types b as sets of constants.

To define the semantics of types, we first define the following predicate :

Definition 2.1 (*Typing relation*). The predicate $(d' : \tau)$, where d' is either Ω or an element d of $\mathcal{D}$ and τ is a type, is defined recursively in the following way:

$$
\begin{aligned}
(\Omega : \tau) &= \mathsf{false} \\
(c : b) &= c \in \mathbb{B}[\![b]\!] \\
((d_1, d_2) : \tau_1 \times \tau_2) &= (d_1 : \tau_1) \wedge (d_2 : \tau_2) \\
([] : \tau_1 \to \tau_2) &= \mathsf{true} \\
(((d, d') :: \mathrm{f}) : \tau_1 \to \tau_2) &= ((d : \tau_1) \Rightarrow (d' : \tau_2)) \wedge (\mathrm{f} : \tau_1 \to \tau_2) \\
(d : \tau_1 \vee \tau_2) &= (d : \tau_1) \vee (d : \tau_2) \\
(d : \neg\tau) &= \neg(d : \tau) && (d \neq \Omega) \\
(d : \mu v.\tau) &= (d : \tau\{^{\mu v.\tau}/_{v}\}) \\
(d : \tau) &= \mathsf{false} \text{ in any other case}
\end{aligned}
$$

LEMMA 2.2. *This definition is well founded.*

PROOF. To prove that this definition is well founded, we define a structural ordering relation $\trianglelefteq$ on $(\mathcal{D} \cup \{\Omega\}) \times T$ where T is the set of types:

— On $\mathcal{D} \cup \{\Omega\}$, 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.

It is now 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. □

Example 2.3. Consider the value $d = (2, \texttt{nil})$ where 2 has base type $\texttt{int}$ and $\texttt{nil}$ is the only value of type $\texttt{nil}$. Let $\tau = \mu v.\texttt{nil} \vee (\texttt{int} \times v)$. We can compute $(d : \tau)$ in the following way:

— Unfolding: $(d : \tau) = (d : \texttt{nil} \vee (\texttt{int} \times \tau))$
— Disjunction: $d \notin \mathcal{C}$ therefore $(d : \texttt{nil}) = \mathsf{false}$ since $\texttt{nil}$ is a base type.
Thus, $(d : \tau) = (d : \texttt{int} \times \tau)$.
— Pair deconstruction: $(2 : \texttt{int}) = \mathsf{true}$ (since $2 \in \mathbb{B}[\![\texttt{int}]\!]$), therefore $(d : \tau) = (\texttt{nil} : \tau)$
— Unfolding: $(d : \tau) = (\texttt{nil} : \texttt{nil} \vee (\texttt{int} \times \tau))$
— Disjunction: $(\texttt{nil} : \texttt{nil}) = \mathsf{true}$, therefore $(d : \tau) = \mathsf{true}$.

Definition 2.4 (*Interpretation of types*). The interpretation of types as parts of $\mathcal{D}$ is defined as $[\![\tau]\!] = \{d \mid (d : \tau)\}$. Note that Ω is not part of any type, as expected.

We can now give the standard definition of semantic subtyping.

Definition 2.5 (*Subtyping*). 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 described by Genevès et al. [2015]: a variant of μ-calculus whose models are finite binary trees.

Data model. Let Σ be a set of *labels*, ranged over by ζ. We consider binary trees where each node bears a finite number of labels; we use L to range over finite sets of labels. The syntax of our data model is as follows.

$$\begin{array}{rcll} t & ::= & (L, st, st) & \text{binary tree} \\ st & ::= & t \quad | \quad \epsilon & \text{subtree} \end{array}$$

We write $\mathcal{T}$ for the set of trees generated by the above grammar.

Logic formulas. The logic language we use allows describing properties of such trees. In addition to standard Boolean connectives, it comprises *atomic propositions* ζ, indicating that the label ζ is present at the root; *existential modalities* $\langle 1 \rangle$ and $\langle 2 \rangle$ stating the existence of a non-empty left (first) or right (second) subtree, respectively, satisfying some formula; and a polyadic fixpoint binder μ to express recursion.

We use the letter a to range over $\{1, 2\}$, the two possible directions of navigation.

The syntax of formulas is formally defined as follows:

$$\begin{array}{rcll} \varphi, \psi & ::= & & \text{formula} \\ & & \top \quad | \quad \bot & \text{true, false} \\ & | & \zeta \quad | \quad \neg\zeta & \text{atomic proposition (negated)} \\ & | & X & \text{fixpoint variable} \\ & | & \varphi \vee \psi & \text{disjunction} \\ & | & \varphi \wedge \psi & \text{conjunction} \\ & | & \langle a \rangle\, \varphi \quad | \quad \neg \langle a \rangle\, \top & \text{existential modality (negated)} \\ & | & \mu(X_i = \varphi_i)_{i \in I} \text{ in } \psi & \text{(least) polyadic fixpoint} \end{array}$$

The polyadic fixpoint is a way of defining a set of mutually recursive formulas. As an additional constraint, we require this recursion to be *guarded* by modalities, i. e. the formula $\mu(X_i = \varphi_i)_{i \in I}$ in ψ must be such that all occurrences of the X_i in the φ_j appear in subformulas starting with a modality $\langle a \rangle$.

$$
\begin{aligned}
[\![\top]\!]_V &= \mathcal{T} & [\![\zeta]\!]_V &= \{(L, st, st) \mid \zeta \in L\} \\
[\![\bot]\!]_V &= \emptyset & [\![\neg\zeta]\!]_V &= \{(L, st, st) \mid \zeta \notin L\} \\
[\![\varphi \vee \psi]\!]_V &= [\![\varphi]\!]_V \cup [\![\psi]\!]_V & [\![\varphi \wedge \psi]\!]_V &= [\![\varphi]\!]_V \cap [\![\psi]\!]_V \\
[\![\langle \mathbf{1} \rangle\, \varphi]\!]_V &= \{(L, t, st) \mid t \in [\![\varphi]\!]_V\} & [\![\neg \langle \mathbf{1} \rangle\, \top]\!]_V &= \{(L, \epsilon, st)\} \\
[\![\langle \mathbf{2} \rangle\, \varphi]\!]_V &= \{(L, st, t) \mid t \in [\![\varphi]\!]_V\} & [\![\neg \langle \mathbf{2} \rangle\, \top]\!]_V &= \{(L, st, \epsilon)\} \\
[\![X]\!]_V &= V(X)
\end{aligned}
$$

$[\![\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V = [\![\psi]\!]_{V[\overline{X_i \mapsto U_i}]}$ where the U_i are defined as follows:

$$
\begin{aligned}
&\text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{T})^I \mid \forall j \in I, [\![\varphi_j]\!]_{V[\overline{X_i \mapsto T_i}]} \subseteq T_j\} \\
&\text{and for all } j \in I, \text{ let } U_j = \bigcap_{(T_i)\in S} T_j
\end{aligned}
$$

where $V[\overline{X_i \mapsto T_i}](X) = \begin{cases} T_k & \text{if } X = X_k \\ V(X) & \text{otherwise.} \end{cases}$

Fig. 1. Interpretation of formulas

Interpretation of formulas. The interpretation of formulas as subsets of $\mathcal{T}$ is defined on Figure 1, where V is a valuation, i. e. a mapping from fixpoint variables to subsets of $\mathcal{T}$. When φ is closed, its interpretation is independent of V; in that case, we simply write it $[\![\varphi]\!]$.

The interpretation of fixpoint formulas may seem complicated. It is, however, standard and corresponds to a least prefixpoint; it allows giving a definition before having proved that a fixpoint exists and is unique. In our case, with this logic language and this interpretation, we actually know that there is a unique fixpoint. This is expressed formally by the following property, which we will use instead of the definition:

PROPERTY 3.1. *Let $\mu(X_i = \varphi_i)_{i\in I}$ in ψ be a fixpoint formula. Then there exists a unique tuple $(U_i)_{i\in I}$ of parts of $\mathcal{T}$ such that: $\forall j \in I, [\![\varphi_j]\!]_{V[\overline{X_i \mapsto U_i}]} = U_j$. We call $\overline{X_i \mapsto U_i}$ the* fixpoint valuation *of the formula, and we have $[\![\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V = [\![\psi]\!]_{V[\overline{X_i \mapsto U_i}]}$.*

This property is a consequence of a more general result from Genevès et al. [2015] on *cycle-free* formulas. The cycle-freeness constraint defined in that paper translates as the guarded-recursion constraint in the sublanguage we use here. We refer the reader to that paper for the proof.

Syntactic sugar. In addition to what the syntax generates, we use some abbreviations. We write $\mu X.\varphi$ for $\mu(X = \varphi)$ in X. The *universal modalities* $[a]$ are defined by : $[a]\,\varphi = \neg \langle a \rangle\, \top \vee \langle a \rangle\, \varphi$ ('If there is an a subtree, then it satisfies φ'). When φ is a closed formula, $\neg\varphi$ can also be defined as an abbreviation for the following $\mathrm{neg}(\cdot)$ operation :

Definition 3.2 (*Negation of a formula*). The negation of a formula in general (possibly containing free variables) is *not* defined. To define it for closed formulas, we use the following $\mathrm{neg}(\cdot)$ operation. This operation is defined inductively on all formulas; however, it only represents negation when its argument is a closed formula.

$$
\begin{aligned}
\mathrm{neg}(\zeta) &= \neg\zeta & \mathrm{neg}(\langle a \rangle\, \varphi) &= [a]\, \mathrm{neg}(\varphi) \\
\mathrm{neg}(\top) &= \bot & \mathrm{neg}(\bot) &= \top \\
\mathrm{neg}(\varphi \vee \psi) &= \mathrm{neg}(\varphi) \wedge \mathrm{neg}(\psi) & \mathrm{neg}(\varphi \wedge \psi) &= \mathrm{neg}(\varphi) \vee \mathrm{neg}(\psi) \\
\mathrm{neg}(\neg\varphi) &= \varphi & \mathrm{neg}(X) &= X
\end{aligned}
$$

$$\mathrm{neg}(\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi) = \mu(X_i = \mathrm{neg}(\varphi_i))_{i\in I} \text{ in } \mathrm{neg}(\psi)$$

PROPERTY 3.3. *If φ is a closed formula,* $[\![\mathrm{neg}(\varphi)]\!] = \mathcal{T} \setminus [\![\varphi]\!]$.

PROOF. For any valuation V, we define its complement valuation $\neg V$ by $(\neg V)(X) = \mathcal{T} \setminus V(X)$. We can then prove by a structural induction that for any formula ψ and any V we have $[\![\mathrm{neg}(\psi)]\!]_V = \mathcal{T} \setminus [\![\psi]\!]_{\neg V}$. The induction is straightforward (it relies on Property 3.1 in the fixpoint case). The property we stated is then just the particular case where ψ is closed. □

When φ is closed, we write $\neg\varphi$ for $\mathrm{neg}(\varphi)$.

4. LOGICAL ENCODING

In the context of the present paper, we want finite tree models of the logic to correspond to the domain elements d introduced in Section 2, so that we can associate to each type a formula whose set of models corresponds to the denotation of the type. Thus, we first choose an appropriate alphabet Σ of node labels and a representation of domain elements. Then, we present the translation of a type into a logical formula.

4.1. Representation of domain elements

We now describe a way to represent elements of our semantic domain $\mathcal{D}$ as trees, so that we can reason on $\mathcal{D}$ using our tree logic. The first step is to represent constants from $\mathcal{C}$; but $\mathcal{C}$ is arbitrary in the semantic subtyping framework. However, our purpose is just to decide subtyping; for this purpose, it is not useful to distinguish values which belong to exactly the same types. We therefore associate to each basic type b a tree label $\mathsf{lbl}(b)$, and associate to each constant from $\mathcal{C}$ a leaf node bearing the labels corresponding to all the basic types it belongs to:

$$\mathsf{tree}(c) = (\{\mathsf{lbl}(b) \mid c \in \mathbb{B}[\![b]\!]\}, \epsilon, \epsilon).$$

Now that we have a representation of constants as leaves, we associate to each type constructor a specific tree label: $\textcircled{\rightarrow}$ for arrow and $\textcircled{\times}$ for product, to which we add $\textcircled{\Omega}$ for the error value Ω. We suppose all the $\mathsf{lbl}(b)$ are different from these three labels and from each other. If we assume Σ comprises all these labels, we can define the following injective translation tree from $\mathcal{D} \cup \{\Omega\} \cup (\mathcal{D} \times (\mathcal{D} \cup \{\Omega\}))$ to $\mathcal{T}$:

$$\begin{aligned}
\mathsf{tree}(c) &= (\{\mathsf{lbl}(b) \mid c \in \mathbb{B}[\![b]\!]\}, \epsilon, \epsilon)\\
\mathsf{tree}(\Omega) &= (\{\textcircled{\Omega}\}, \epsilon, \epsilon)\\
\mathsf{tree}((d, d')) &= (\{\textcircled{\times}\}, \mathsf{tree}(d), \mathsf{tree}(d'))\\
\mathsf{tree}([]) &= (\{\textcircled{\rightarrow}\}, \epsilon, \epsilon)\\
\mathsf{tree}(((d, d') :: \mathrm{f})) &= (\{\textcircled{\rightarrow}\}, \mathsf{tree}((d, d')), \mathsf{tree}(\mathrm{f}))
\end{aligned}$$

The intuition of this tree representation is illustrated in Figure 2.

4.2. Translation of types

We now assume that $\Sigma = \{\mathsf{lbl}(b) \mid b \in \mathcal{B}\} \cup \{\textcircled{\Omega}, \textcircled{\rightarrow}, \textcircled{\times}\}$.

Now that domain values are represented by trees, we can represent types, which denote sets of domain values, with formulas which denote sets of trees. But as a preliminary, since not all trees are the representation of a domain value, we need a formula describing exactly the image of $\mathcal{D}$ by function tree, so that we can easily exclude trees corresponding to nothing.

This first requires a formula which describes the image of $\mathcal{C}$; let us call this formula isbase. The actual formula depends on the language considered, and more precisely on the set-theoretic relations between the basic types' denotations. Indeed, we know that

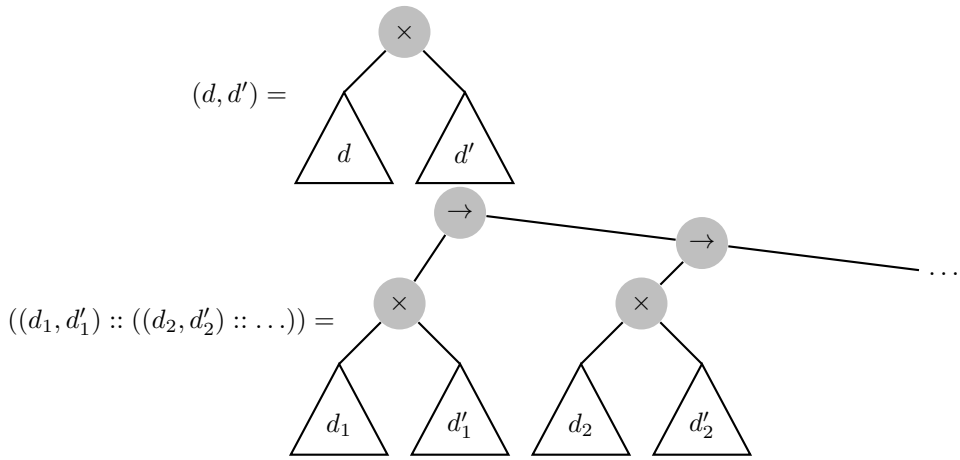

Fig. 2. Pairs and functions are represented as trees with special labels.

a basic constant is represented by a leaf which bears only labels of the form $\mathsf{lbl}(b)$, but additionally, we need to know which combinations of basic types are possible (i.e. have a non-empty denotation) and which are not.

The semantic subtyping framework is parametric on the set of constants $\mathcal{C}$ and the set-theoretic interpretation $\mathbb{B}[\![\cdot]\!]$ of basic types; but what really matters in these parameters with respect to the subtyping relation is: among all intersections of basic types and negations of basic types, which ones are empty and which ones are not? Let basic_constraint be a Boolean formula of atomic propositions $\mathsf{lbl}(b)$ which describes exactly the set of possible combinations of basic types, as determined by $\mathbb{B}[\![\cdot]\!]$. This formula is the only parameter of the logical translation of the framework.

Formally, this formula is linked to $\mathcal{C}$ and $\mathbb{B}[\![\cdot]\!]$ in the following way:

Definition 4.1. basic_constraint is a formula comprising only atomic propositions and Boolean connectors such that:

$$[\![\mathsf{basic_constraint}]\!] = \{(L, st_1, st_2) \in \mathcal{T} \mid \exists c \in \mathcal{C}, \forall b \in \mathcal{B}, \mathsf{lbl}(b) \in L \Leftrightarrow c \in \mathbb{B}[\![b]\!]\}$$

Since $\mathcal{B}$ is finite, such a formula always exists: it can be obtained by enumerating all non-empty combinations of basic types.

In practice, in programming languages, two basic types are usually either in a subtyping relation or completely disjoint (which corresponds to a subtyping relation between one and the negation of the other). This can be encoded by a conjunction of clauses which relate two consecutive types, together with one clause to ensure every constant has at least one basic type.

As an example, suppose the basic types are `integer`, `natural` and `string`. The usual relations between these types are: `natural` is a (strict) subtype of `integer`, and `string` is completely disjoint. Let $s = \mathsf{lbl}(\texttt{string}), n = \mathsf{lbl}(\texttt{natural}), i = \mathsf{lbl}(\texttt{integer})$. In this case the formula would be: $\mathsf{basic_constraint} = (\neg n \vee i) \wedge (\neg s \vee \neg i) \wedge (s \vee i)$. We could also imagine that we use a language where automatic conversion occurs, as needed, between strings and integers. Since there are strings which do not represent integers, but any integer can be represented by a string (and thus, in our small example, all constants can be represented by strings), we would just change the formula into $\mathsf{basic_constraint} = (\neg n \vee i) \wedge s$. If we add a type `alphabetic` limited to strings comprising only letters, the type `string` would then have two disjoint subtypes, which we could encode as $\mathsf{basic_constraint} = (\neg n \vee i) \wedge (\neg i \vee \neg a) \wedge s$.

$$
\begin{aligned}
\mathsf{notbase} &= \bigwedge_{b \in \mathcal{B}} \neg \mathsf{lbl}(b) \\
\mathsf{isleaf} &= \neg \langle \mathbf{1} \rangle \top \wedge \neg \langle \mathbf{2} \rangle \top \\
\mathsf{prod} &= \otimes \wedge \neg \rightarrow \wedge \neg \Omega \wedge \mathsf{notbase} \\
\mathsf{arrow} &= \rightarrow \wedge \neg \otimes \wedge \neg \Omega \wedge \mathsf{notbase} \\
\mathsf{error} &= \Omega \wedge \neg \rightarrow \wedge \neg \otimes \wedge \mathsf{notbase} \wedge \mathsf{isleaf} \\
\mathsf{isbase} &= \mathsf{basic_constraint} \wedge \neg \Omega \wedge \neg \rightarrow \wedge \neg \otimes \wedge \mathsf{isleaf} \\
\mathsf{isd} &= \mu X.(\\
&\qquad \mathsf{isbase} \vee \\
&\qquad (\mathsf{prod} \wedge \langle \mathbf{1} \rangle X \wedge \langle \mathbf{2} \rangle X) \vee \\
&\qquad \mu Y.(\mathsf{arrow} \wedge (\\
&\qquad\quad \mathsf{isleaf} \vee \\
&\qquad\quad (\langle \mathbf{1} \rangle (\mathsf{prod} \wedge \langle \mathbf{1} \rangle X \wedge \langle \mathbf{2} \rangle (X \vee \mathsf{error})) \wedge \langle \mathbf{2} \rangle Y) \\
&\qquad)) \\
&\quad)
\end{aligned}
$$

Fig. 3. Formulas describing different parts of the image set of function tree

From the formula basic_constraint, we can define the formulas of Figure 3 to describe the image of $\mathcal{D}$. The first formulas are used to enforce the labelling constraints (special labels are mutually exclusive and incompatible with the base types). Then isbase selects all trees corresponding to constants from $\mathcal{C}$. error is straightforward. isd selects all elements of $\mathcal{D}$: either they are a constant, or a pair (a $\otimes$ node with exactly two children, each of which is itself in $\mathcal{D}$), or a function: a $\rightarrow$ node with either no children at all or a first child which is a pair whose second element may be error and a second child which is itself a function. Altogether, we have the following lemma:

LEMMA 4.2. $[\![\mathsf{isd}]\!] = \{\mathsf{tree}(d) \mid d \in \mathcal{D}\}$.

PROOF. Using the definition of the interpretation of formulas, we have that:

(1) $[\![\mathsf{notbase}]\!] = \{(L, st_1, st_2) \in \mathcal{T} \mid L \subseteq \{\Omega, \rightarrow, \otimes\}\}$ (from how Σ was defined);
(2) $[\![\mathsf{isleaf}]\!] = \{(L, \epsilon, \epsilon) \in \mathcal{T}\}$;
(3) $[\![\mathsf{prod}]\!] = \{(\{\otimes\}, st_1, st_2) \in \mathcal{T}\}$ (from (1));
(4) $[\![\mathsf{arrow}]\!] = \{(\{\rightarrow\}, st_1, st_2) \in \mathcal{T}\}$ (from (1));
(5) $[\![\mathsf{isbase}]\!] = \{\mathsf{tree}(c) \mid c \in \mathcal{C}\}$ (from Def. 4.1 and (2));
(6) $[\![\mathsf{error}]\!] = \{\mathsf{tree}(\Omega)\}$ (from (1) and (2)).

Then let $U = \{\mathsf{tree}(d) \mid d \in \mathcal{D}\}$ and $U' = \{\mathsf{tree}(\mathrm{f}) \mid \mathrm{f} \in \mathcal{D} \text{ is a function }\}$. Let $V = \{X \mapsto U, Y \mapsto U'\}$. Let ψ and φ be the subformulas respectively bound to X and Y in isd. We can check, from all the preceding, that:

— $[\![\mathsf{arrow} \wedge \mathsf{isleaf}]\!] = \{\mathsf{tree}([])\}$;
— $[\![\mathsf{prod} \wedge \langle \mathbf{1} \rangle X \wedge \langle \mathbf{2} \rangle (X \vee \mathsf{error})]\!]_V = \{\mathsf{tree}((d, d')) \mid d \in \mathcal{D} \text{ and } d' \in \mathcal{D} \cup \{\Omega\}\}$;
— therefore, $[\![\varphi]\!]_V = U'$ (from the two cases of the definition of tree for abstract functions);
— hence U' is the fixpoint valuation for Y, and thus $[\![\mu Y.\varphi]\!]_{\{X \mapsto U\}} = U'$ (from Prop. 3.1);
— in ψ, the other two terms in the disjunction correspond respectively to the set of all base constants (see (5) above) and to the set of all pairs (d, d') such that both d and d' are in $\mathcal{D}$;

$$
\begin{aligned}
\mathsf{form}(b) &= \mathsf{lbl}(b) \\
\mathsf{form}(\tau_1 \times \tau_2) &= \otimes \wedge \langle \mathbf{1} \rangle\, \mathsf{form}(\tau_1) \wedge \langle \mathbf{2} \rangle\, \mathsf{form}(\tau_2) \\
\mathsf{form}(\tau_1 \to \tau_2) &= \mu Y.(\ominus \wedge [\mathbf{2}]\, Y \wedge \\
&\qquad [\mathbf{1}]\, (\langle \mathbf{1} \rangle\, \mathsf{negform}(\tau_1) \vee \langle \mathbf{2} \rangle\, (\neg \Omega \wedge \mathsf{form}(\tau_2))) \\
&\quad) \\
\mathsf{form}(\tau_1 \vee \tau_2) &= \mathsf{form}(\tau_1) \vee \mathsf{form}(\tau_2) \\
\mathsf{form}(\neg \tau) &= \mathsf{negform}(\tau) \\
\mathsf{form}(\mathbf{0}) &= \bot \\
\mathsf{form}(v) &= X_v^+ \\
\mathsf{form}(\mu v.\tau) &= \mu(X_v^+ = \mathsf{form}(\tau), X_v^- = \mathsf{negform}(\tau)) \text{ in } X_v^+ \\
\\
\mathsf{negform}(b) &= \neg \mathsf{lbl}(b) \\
\mathsf{negform}(\tau_1 \times \tau_2) &= \neg \otimes \vee \langle \mathbf{1} \rangle\, \mathsf{negform}(\tau_1) \vee \langle \mathbf{2} \rangle\, \mathsf{negform}(\tau_2) \\
\mathsf{negform}(\tau_1 \to \tau_2) &= \mu Y.(\neg \ominus \vee \langle \mathbf{2} \rangle\, Y \vee \\
&\qquad \langle \mathbf{1} \rangle\, ([\mathbf{1}]\, \mathsf{form}(\tau_1) \wedge [\mathbf{2}]\, (\Omega \vee \mathsf{negform}(\tau_2))) \\
&\quad) \\
\mathsf{negform}(\tau_1 \vee \tau_2) &= \mathsf{negform}(\tau_1) \wedge \mathsf{negform}(\tau_2) \\
\mathsf{negform}(\neg \tau) &= \mathsf{form}(\tau) \\
\mathsf{negform}(\mathbf{0}) &= \top \\
\mathsf{negform}(v) &= X_v^- \\
\mathsf{negform}(\mu v.\tau) &= \mu(X_v^+ = \mathsf{form}(\tau), X_v^- = \mathsf{negform}(\tau)) \text{ in } X_v^-
\end{aligned}
$$

Fig. 4. Translation from types to formulas.

— therefore, $[\![\psi]\!]_{\{X \mapsto U\}} = U$, thus U is the fixpoint valuation for X and $[\![\mu X.\psi]\!] = U$. $\square$

We can now define formulas corresponding to the types themselves. The type language allows recursion variables to appear in contravariant positions, which is not permitted in the μ-calculus. To cater for that, we translate recursive types with a pair of mutually recursive formulas, one representing the type and the other its negation. Note that in most cases, the formulas will actually not be mutually recursive and only one of the two will be used; it is however simpler to define a translation which works in all cases and can be simplified afterwards by removing unused subformulas than to distinguish particular cases in the definition.

To each recursion variable v, we associate a pair of μ-calculus fixpoint variables X_v^+ and X_v^-, which are all distinct from each other and from Y. We then associate to every type τ the formula $\mathsf{fullform}(\tau) = \mathsf{isd} \wedge \mathsf{form}(\tau)$, with $\mathsf{form}(\tau)$ defined in Fig. 4.

The translation of product types is simple: it describes a $\otimes$ node whose first child is described by $\mathsf{form}(\tau_1)$ and whose second child is described by $\mathsf{form}(\tau_2)$. The translation of arrow types describes a $\ominus$ node whose right child, if it exists, has the same structure as itself recursively and whose left child, if it exists, must be a node with two children such that either the first does not have type τ_1 ($\mathsf{negform}(\tau_1)$) or the second has type τ_2 ($\neg \Omega \wedge \mathsf{form}(\tau_2)$). This is the only place in this translation where we have to specify that

a node must not be labelled $\textcircled{\Omega}$, because everywhere else it is already enforced by isd[3]. On this particular node, given the constraints implied by isd, adding the constraint $\neg\textcircled{\Omega}$ is sufficient to say that it must correspond to an element of $\mathcal{D}$. There are some other constraints which are already enforced by isd and need not be repeated here, like the facts that an arrow node must have exactly zero or two children, or that its left child must have label $\textcircled{\times}$.

Notice that in this translation $\mu v.\tau$ is directly translated as a fixpoint formula with two variables. Since τ might itself contain a recursive type, the size of the obtained formula may not be optimal: in the worst case, it can be exponentially larger than the size of the types. To avoid this problem, one could use an alternative translation using a single polyadic fixpoint for the whole initial type binding all variables at once, as discussed in Section 5.4. Nevertheless, we prefer the above definition for clarity.

LEMMA 4.3. *If τ is closed, then so are* $\mathsf{negform}(\tau)$ *and* $\mathsf{form}(\tau)$*, and we have* $[\![\mathsf{negform}(\tau)]\!] = [\![\neg\mathsf{form}(\tau)]\!]$.

PROOF. By a straightforward induction, we can see that for any (possibly open) type τ, the formulas $\mathsf{negform}(\tau)$ and $\mathrm{neg}(\mathsf{form}(\tau))$ are almost identical, the only difference being that the variables X_v^+ and X_v^- are exchanged for all v. Since replacing every X_v^+ with X_v^- and vice-versa is an injective renaming of variables, it has no effect if all the variables are bound; therefore, in that case the two formulas are equivalent. □

Correctness of the translation. For any closed type τ, we have:

$$[\![\mathsf{fullform}(\tau)]\!] = \{\mathsf{tree}(d) \mid d \in [\![\tau]\!]\}.$$

This property will be formally proved for polymorphic types in the next section (Theorem 5.6). (The proof in the monomorphic case is the same with one less case.)

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. 2015], this means that this translation gives an alternative way to decide the classical semantic subtyping relation as defined by 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 [Milner et al. 1975], not higher-rank polymorphism, so there are no quantifiers in the syntax of types.

[3]A former version of our encoding erroneously omitted $\neg\textcircled{\Omega}$ at this point, because it does not appear explicitly in $(d : \tau_1) \Rightarrow (d' : \tau_2)$, which is what we are translating here. However it does appear *implicitly* in the fact that d' ranges over $\mathcal{D} \cup \{\Omega\}$ and that $(d' : \tau_2)$ is always false if $d' = \Omega$.

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 by 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:

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

Obviously the condition on the right is *necessary* for subtyping to hold. But deciding that it is *sufficient* as well makes the relation unsatisfactory and somehow counterintuitive, as remarked by 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 conforms to the definition because 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], who concentrate on XML types, use 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 by 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 [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 interpretations 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 indeed the semantic subtyping relation on polymorphic types defined by Castagna and Xu [2011]. These steps are formally detailed in the following section.

5.2. Interpretation of polymorphic types

5.2.1. Extended semantic domain. Let Λ be an infinite set of tags, disjoint from $\{\mathsf{lbl}(b) \mid b \in \mathcal{B}\} \cup \{\textcircled{\Omega}, \textcircled{\rightarrow}, \textcircled{\times}\}$, 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 *tags* which tag elements of the semantic domain from *variables* which occur in types.)

In the following, we let λ range over finite parts of Λ. We define the extended semantic domain $\mathcal{D}^*$ by allowing every syntactic node of domain elements to be tagged with an arbitrary λ. Formally, this corresponds to the set of δs generated by the following grammar:

$$
\begin{array}{rcll}
\delta & ::= & & \text{tagged domain element} \\
 & & c_\lambda & \text{tagged base constant} \\
 & | & (\delta, \delta)_\lambda & \text{tagged pair} \\
 & | & f_\lambda & \text{tagged function} \\
f & ::= & & \\
 & & [] & \\
 & | & ((\delta, \delta')_\lambda :: f_\lambda) & \\
\delta' & ::= & & \\
 & & \delta & \\
 & | & \Omega_\lambda & \text{tagged error}
\end{array}
$$

For δ in $\mathcal{D}^*$, let $\mathsf{tags}(\delta)$ be the toplevel tags of δ, i.e.:

$$
\begin{aligned}
\mathsf{tags}(c_\lambda) &= \lambda \\
\mathsf{tags}((\delta_1, \delta_2)_\lambda) &= \lambda \\
\mathsf{tags}(f_\lambda) &= \lambda
\end{aligned}
$$

Remark 5.1. The error Ω may be tagged, and the head and tail of a list representing an abstract function may be tagged separately from the list itself, for regularity reasons; having them untagged would complicate the translation into logic that we will

present in Sec. 5.3. However, since they do not represent domain elements themselves but mere intermediate nodes in the syntactic tree, their tags are actually irrelevant.

5.2.2. Fixed interpretation of polymorphic types. We now define an interpretation of polymorphic types as subsets of $\mathcal{D}^*$, very similarly to what we did for monomorphic types, with just an additional case for the type variable:

Definition 5.2. [Typing relation for polymorphic types] The predicate $(\delta' : \tau)$ is defined recursively in the following way:

$$
\begin{aligned}
(\Omega_\lambda : \tau) &= \mathsf{false} \\
(c_\lambda : b) &= c \in \mathbb{B}[\![b]\!] \\
((\delta_1, \delta_2)_\lambda : \tau_1 \times \tau_2) &= (\delta_1 : \tau_1) \wedge (\delta_2 : \tau_2) \\
([]_\lambda : \tau_1 \to \tau_2) &= \mathsf{true} \\
(((\delta, \delta')_\lambda :: f_{\lambda'}) : \tau_1 \to \tau_2) &= ((\delta : \tau_1) \Rightarrow (\delta' : \tau_2)) \wedge (f_{\lambda'} : \tau_1 \to \tau_2) \\
(\delta : \alpha) &= \iota(\alpha) \in \mathsf{tags}(\delta) \\
(\delta : \tau_1 \vee \tau_2) &= (\delta : \tau_1) \vee (\delta : \tau_2) \\
(\delta : \neg\tau) &= \neg(\delta : \tau) \\
(\delta : \mu v.\tau) &= (\delta : \tau\{^{\mu v.\tau}/_v\}) \\
(\delta : \tau) &= \mathsf{false} \text{ in any other case}
\end{aligned}
$$

With a reasoning similar to the monomorphic case, this definition is well founded. We define the interpretation of a polymorphic type as $(\!|\tau|\!) = \{\delta \in \mathcal{D}^* \mid (\delta : \tau)\}$.

For ground types, $(\!|\tau|\!)$ is very similar to $[\![\tau]\!]$, but 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 taggings for each of its values.

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 value d in the non-extended domain, there always exist both an infinity of tagged 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 extended domain big enough to contain all possible cases.

5.2.3. Equivalence with Castagna and Xu's definition of subtyping. In this section, we prove that our definition of subtyping for polymorphic types, Definition (4) above, is equivalent to the one of Castagna and Xu [2011], and thus accurately represents a relation that holds *independently of the variables*. For this, we have to introduce a few additional notions which Castagna and Xu's framework relies on, notably a different interpretation of types (on the same semantic domain). These notions are not used elsewhere; the development in this section is meant to justify that defining subtyping with (4) is correct, and is quite separate from the rest of our formal development.

We first introduce *assignments* η: functions from $\mathcal{V}$ to $\mathcal{P}(\mathcal{D}^*)$. An assignment attributes to each variable an arbitrary set of elements from the (extended) semantic domain.

We then define the interpretation of a type *relative to an assignment* in the following way: the predicate $(\delta' :_\eta \tau)$ is defined inductively exactly as $(\delta' : \tau)$ except for type variables: where the fixed interpretation is $(\delta : \alpha) = \iota(\alpha) \in \mathsf{tags}(\delta)$, we have instead:

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

The interpretation of the polymorphic type τ relative to the assignment η is then $[\![\tau]\!]\eta = \{\delta \mid (\delta :_\eta \tau)\}$. 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:

$$\eta_\iota(\alpha) = \{\delta \in \mathcal{D}^* \mid \iota(\alpha) \in \mathsf{tags}(\delta)\}.$$

This assignment is such 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 5.3. *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: $\forall \tau \in T, \forall \delta \in \mathcal{D}^*, \delta \in [\![\tau]\!]\eta \Leftrightarrow F_V^\eta(\delta) \in [\![\tau]\!]\eta_\iota$.*

PROOF. For δ in $\mathcal{D}^*$, let $L(\delta) = \{\iota(\alpha) \mid \alpha \in V \wedge \delta \in \eta(\alpha)\}$. Since V is finite, $L(\delta)$ is finite as well. We define $F_V^\eta(\delta)$ inductively; in order to do that, we define it not just on elements of $\mathcal{D}^*$ but on intermediate syntactic nodes as well. On domain elements:

— If $\delta = c_\lambda$ then $F_V^\eta(\delta) = c_{L(\delta)}$
— If $\delta = (\delta_1, \delta_2)_\lambda$ then $F_V^\eta(\delta) = (F_V^\eta(\delta_1), F_V^\eta(\delta_2))_{L(\delta)}$
— If $\delta = f_\lambda$ then $F_V^\eta(\delta) = F_V^\eta(f)_{L(\delta)}$

and on other syntactic nodes:

— $F_V^\eta([]) = []$
— $F_V^\eta(((\delta_1, \delta_1')_{\lambda_1} :: f_{\lambda_2})) = ((F_V^\eta(\delta_1), F_V^\eta(\delta_2))_{\lambda_1} :: F_V^\eta(f)_{\lambda_2})$
— $F_V^\eta(\Omega_\lambda) = \Omega_\lambda$

So F_V^η preserves the structure but changes the tags so that $\mathsf{tags}(F_V^\eta(\delta)) = L(\delta)$ and so on inductively for its subterms. The tags of intermediate nodes are not changed[4].

Let $\mathcal{P}(\delta, \tau) = \delta \in [\![\tau]\!]\eta \Leftrightarrow F_V^\eta(\delta) \in [\![\tau]\!]\eta_\iota$. We prove that this predicate holds for all pairs (δ, τ) such that τ is in T, by induction on those pairs, using the ordering

[4]As said in Remark 5.1, these tags are irrelevant.

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 $\mathcal{P}(\delta, \tau)$ is true by definition of $L(\delta)$.
— If it is a base type. Then $\mathcal{P}(\delta, \tau)$ is true because the interpretation of τ is independent of assignments and of tags.

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

— For the function 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 5.4. *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 exist η and δ such that $\delta \in [\![\tau]\!]\eta$. From the previous lemma we then have $F^\eta_{\mathsf{var}(\tau)}(\delta) \in [\![\tau]\!]\eta_\iota$. □

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). Indeed:

— Def. (4) tells us that $\tau_1 \leqslant \tau_2$ holds iff $(\!|\tau_1|\!) \subseteq (\!|\tau_2|\!)$, i. e. iff $(\!|\tau_1|\!) \setminus (\!|\tau_2|\!) = (\!|\tau_1 \setminus \tau_2|\!) = \emptyset$, and we know that $(\!|\tau_1 \setminus \tau_2|\!) = [\![\tau_1 \setminus \tau_2]\!]\eta_\iota$.

— Def. (5) tells us that $\tau_1 \leqslant \tau_2$ holds iff for all η we have $[\![\tau_1]\!]\eta \subseteq [\![\tau_2]\!]\eta$, i. e. $[\![\tau_1]\!]\eta \setminus [\![\tau_2]\!]\eta = [\![\tau_1 \setminus \tau_2]\!]\eta = \emptyset$. Saying that it is empty for all η is the same as saying that the union is empty: $\bigcup_{\eta \in \mathcal{P}(\mathcal{D}^*)^{\mathcal{V}}} [\![\tau_1 \setminus \tau_2]\!]\eta = \emptyset$.

— Corollary 5.4 applied to $\tau = \tau_1 \setminus \tau_2$ then proves that both are equivalent.

Convexity of the model. Definition (5) corresponds to semantic subtyping as defined by 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 by Castagna and Xu [2011], is a sufficient condition. But we need not even rely on this result since Corollary 5.4 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.

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

5.3. Logical encoding of variables

5.3.1. Representation of extended domain elements. In order to represent tags in our tree language, we simply add to the alphabet Σ of tree labels the set Λ of tags:

$$\Sigma = \{\mathsf{lbl}(b) \mid b \in \mathcal{B}\} \cup \{\textcircled{\Omega}, \textcircled{\rightarrow}, \textcircled{\times}\} \cup \Lambda$$

The translation of extended domain elements into trees is then similar to what we had in the monomorphic case; we simply add the encoding of labels:

$$\begin{aligned}
\mathsf{treex}(c_\lambda) &= (\lambda \cup \{\mathsf{lbl}(b) \mid c \in \mathbb{B}[\![b]\!]\}, \epsilon, \epsilon)\\
\mathsf{treex}(\Omega_\lambda) &= (\lambda \cup \{\textcircled{\Omega}\}, \epsilon, \epsilon)\\
\mathsf{treex}((\delta, \delta')_\lambda) &= (\lambda \cup \{\textcircled{\times}\}, \mathsf{treex}(\delta), \mathsf{treex}(\delta'))\\
\mathsf{treex}([]_\lambda) &= (\lambda \cup \{\textcircled{\rightarrow}\}, \epsilon, \epsilon)\\
\mathsf{treex}(((\delta, \delta')_{\lambda_1} :: f_{\lambda_2})_\lambda) &= (\lambda \cup \{\textcircled{\rightarrow}\}, \mathsf{treex}((\delta, \delta')_{\lambda_1}), \mathsf{treex}(f_{\lambda_2}))
\end{aligned}$$

5.3.2. Representation of polymorphic types. The translation of types is the same as in the monomorphic case; we just add an additional case for the type variable. In particular, the formula isd is completely unchanged: the passage from $\mathcal{D}$ to $\mathcal{D}^*$ is simply a consequence of the fact we extended Σ to include tags. Since isd contains no constraint about these new labels, it means that every node of every tree in $[\![\mathsf{isd}]\!]$ can bear any number of them. Formally, we can easily adapt the proof of Lemma 4.2 to obtain:

LEMMA 5.5. $[\![\mathsf{isd}]\!] = \{\mathsf{treex}(\delta) \mid \delta \in \mathcal{D}^*\}$

The differences come from the fact that we now have:

(1) $[\![\mathsf{notbase}]\!] = \{(L, st_1, st_2) \in \mathcal{T} \mid L$ is a finite subset of $\{\textcircled{\Omega}, \textcircled{\rightarrow}, \textcircled{\times}\} \cup \Lambda\}$ (because Σ now contains Λ), and
(5) $[\![\mathsf{isbase}]\!] = \{\mathsf{treex}(c_\lambda) \mid c \in \mathcal{C}$ and λ is a finite subset of $\Lambda\}$, for the same reason.

These differences propagate to the rest of the reasoning, which is otherwise unchanged.

The translation $\mathsf{form}(\tau)$ of types into formulas is extended by translating each type variable into the atomic proposition denoting the tag associated with the variable:

$$\begin{aligned}
\mathsf{form}(\alpha) &= \iota(\alpha)\\
\mathsf{negform}(\alpha) &= \neg\iota(\alpha)
\end{aligned}$$

The other cases are unchanged; in particular, the translation of a ground type is exactly the same formula as in the monomorphic case. We then again define $\mathsf{fullform}(\tau) = \mathsf{isd} \wedge \mathsf{form}(\tau)$, and we have the following result:

THEOREM 5.6. *For any closed type τ,* $[\![\mathsf{fullform}(\tau)]\!] = \{\mathsf{treex}(\delta) \mid \delta \in (\!|\tau|\!)\}$.

PROOF. First of all, Lemma 5.5 allows us to reformulate the statement as: $[\![\mathsf{form}(\tau)]\!] \cap \{\mathsf{treex}(\delta) \mid \delta \in \mathcal{D}^*\} = \{\mathsf{treex}(\delta) \mid \delta \in (\!|\tau|\!)\}$. Using the definition of $(\!|\cdot|\!)$ in terms of the predicate $(\delta : \tau)$, we can further reformulate as follows:

For any closed type τ and any $\delta \in \mathcal{D}^*$, $(\delta : \tau)$ holds if and only if $\mathsf{treex}(\delta) \in [\![\mathsf{form}(\tau)]\!]$.

We prove this result by induction on the pair (δ, τ), using the ordering relation $\trianglelefteq$. In other words, in order to prove that the result is true for a given pair (δ, τ), we assume it is true for all pairs (δ', τ') such that either δ' is a strict subterm of δ, or $\delta = \delta'$ and τ' has a strictly lower shallow depth than τ. We distinguish cases on the form of τ.

— $\tau = b$: $[\![\mathsf{form}(b)]\!]$ contains exactly all leaves which do not have any of the labels $\textcircled{\rightarrow}$, $\textcircled{\Omega}$ and $\textcircled{\times}$ and have label $\mathsf{lbl}(b)$.
If δ is not of the form c_λ, then $(\delta : b)$ does not hold. We also have that $\mathsf{treex}(\delta)$ bears either label $\textcircled{\rightarrow}$ or $\textcircled{\times}$ at the root, thus $\mathsf{treex}(\delta) \notin [\![\mathsf{form}(b)]\!]$.
If $\delta = c_\lambda$, let $\mathsf{treex}(c_\lambda) = (L, \epsilon, \epsilon)$. We have $\mathsf{treex}(c_\lambda) \in [\![\mathsf{form}(b)]\!]$ iff $\mathsf{lbl}(b) \in L$. By definition of treex, this is the case iff $c \in \mathbb{B}[\![b]\!]$, i.e. iff $(c_\lambda : b)$ holds.

— $\tau = \tau_1 \times \tau_2$: if δ is not of the form $(\delta_1, \delta_2)_\lambda$, then $(\delta : \tau)$ does not hold, and the root node of $\mathsf{treex}(\delta)$ cannot have label $\textcircled{\times}$. Therefore we also do not have $\mathsf{treex}(\delta) \in [\![\mathsf{form}(\tau)]\!]$.
If $\delta = (\delta_1, \delta_2)_\lambda$, then we can see that the constraints on δ_1 and δ_2 imposed by $(\delta : \tau)$ and those on $\mathsf{treex}(\delta_1)$ and $\mathsf{treex}(\delta_2)$ imposed by $\mathsf{treex}(\delta) \in [\![\mathsf{form}(\tau)]\!]$ match, and conclude by induction hypothesis.

— $\tau = \tau_1 \to \tau_2$: if δ is not a function, then $(\delta : \tau)$ cannot hold; and since the root of $\mathsf{treex}(\delta)$ does not have label $\textcircled{\rightarrow}$, we do not have $\mathsf{treex}(\delta) \in [\![\mathsf{form}(\tau)]\!]$ either.
If $\delta = []_\lambda$, then $(\delta : \tau)$ is true, and we also have $\mathsf{treex}(\delta) \in [\![\mathsf{form}(\tau)]\!]$ since the root node has label $\textcircled{\rightarrow}$ and no children: the universal modalities in $\mathsf{form}(\tau)$ are satisfied by default.
If $\delta = ((\delta_1, \delta_2')_{\lambda_1} :: f_{\lambda_2})_\lambda$, then $\mathsf{treex}(\delta)$ is a tree with label $\textcircled{\rightarrow}$ at the root and two children. We can see that the constraints on the subtrees expressed by the formula $\mathsf{form}(\tau)$ match the constraints of $(\delta : \tau)$, and we can conclude using the induction hypothesis and Lemma 4.3.

— $\tau = \tau_1 \vee \tau_2$: the result is immediate from the induction hypothesis.

— $\tau = \neg\tau'$: the result is immediate from the induction hypothesis and Lemma 4.3.

— $\tau = \mathbf{0}$: the result is immediate from the definitions.

— $\tau = \alpha$: the result is immediate from the definitions.

— $\tau = \mu v.\tau'$: what we need to prove is that $[\![\mathsf{form}(\tau'\{^\tau/_v\})]\!] = [\![\mathsf{form}(\tau)]\!]$; then the result is immediate from the induction hypothesis.
We have $\mathsf{form}(\tau) = \mu(X_v^+ = \mathsf{form}(\tau'), X_v^- = \mathsf{negform}(\tau'))$ in X_v^+.
Let $V = \{X_v^+ \mapsto U^+, X_v^- \mapsto U^-\}$ be the fixpoint valuation for this formula. We have $U^+ = [\![\mathsf{form}(\tau)]\!]$.
Moreover, we have $\mathsf{negform}(\tau) = \mu(X_v^+ = \mathsf{form}(\tau'), X_v^- = \mathsf{negform}(\tau'))$ in X_v^-. Since the fixpoint binding is the same, the fixpoint valuation is also the same, i.e. V; thus we have $U^- = [\![\mathsf{negform}(\tau)]\!]$.
The difference between $\mathsf{form}(\tau')$ and $\mathsf{form}(\tau'\{^\tau/_v\})$ is that, in the latter, every free occurrence of X_v^+ has been replaced with $\mathsf{form}(\tau)$ and every free occurrence of X_v^- with $\mathsf{negform}(\tau)$. Therefore, from what we showed about V, we have $[\![\mathsf{form}(\tau')]\!]_V = [\![\mathsf{form}(\tau'\{^\tau/_v\})]\!]$. But since V is the fixpoint valuation, we also have $[\![\mathsf{form}(\tau')]\!]_V = U^+$, and we know that $U^+ = [\![\mathsf{form}(\tau)]\!]$. Hence, $[\![\mathsf{form}(\tau)]\!] = [\![\mathsf{form}(\tau'\{^\tau/_v\})]\!]$. □

COROLLARY 5.7. *$\tau_1 \leqslant \tau_2$ holds if and only if* $\mathsf{fullform}(\tau_1 \wedge \neg\tau_2)$*, or alternatively* $\mathsf{isd} \wedge \mathsf{form}(\tau_1) \wedge \mathsf{negform}(\tau_2)$*, is unsatisfiable.*

5.4. Complexity

LEMMA 5.8. *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 is performed by the function $\mathsf{form}(\tau)$. This function does not involve duplication of subformulas of variable size except for the case $\tau = \mu v.\tau'$. Notice that in this case, a naïve implementation of $\mathsf{form}(\tau)$ might produce exponentially-long formulas for nested recursive types (because τ' occurs twice in the translated formula). However, it is easy to generate a formula equivalent to $\mathsf{form}(\tau)$ but of linear size with respect to $|\tau|$ using a single polyadic fixpoint instead of nested fixpoints. In other words, we replace nested recursion with mutual recursion. This is done by modifying the translation in the following ways:

— before translating, we rename all variables in τ so that their names are unique;
— $\mathsf{form}(\mu v.\tau)$ is now X_v^+ and $\mathsf{negform}(\mu v.\tau)$ is now X_v^-;
— we remember the bindings for each variable while processing the formula;
— the result is wrapped in a single top-level polyadic fixpoint that refers to all the variables.

This means that each variable binding is still translated twice, once by form and once by $\mathsf{negform}$, but since all variables are now bound at the top level, none of them is translated *more* than twice.

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 by Genevès et al. [2015] 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 described by Genevès et al. [2015]. 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 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 a leaf or the 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.

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 tries 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 by Genevès et al. [2015]. 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. If 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 one proved subtree for each 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 by Genevès [2006]). This allows our implementation to use 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 counterexamples. 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 reconstruction 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 $\psi = \mathsf{isd} \wedge \mathsf{form}(\tau_1) \wedge \mathsf{negform}(\tau_2)$. 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 counterexample 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 reasons why the relation does not hold.

In the polymorphic setting, a counterexample is in principle, according to the semantics, a labelled tree. However, as mentioned in Section 5.2, whenever a formula is satisfiable there always exist 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 a set of labellings which make that particular tree a counterexample[5]. Namely, for each variable α, every node will be labelled with α to indicate that its set of labels must include α, with $\neg\alpha$ to indicate that it must not, or with nothing if label α is irrelevant for that particular node. This allows an easier interpretation of the counterexample 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://tyrex.inria.fr/websolver/

Concrete Syntax for the 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:

	Paper Syntax	Implementation Syntax
Type variables	α, β, γ	`_a`, `_b`, `_g`
Basic types	b	`_B`
Type constructors	$\times, \rightarrow$	`*`, `->`
Recursion variable	v	`$v`
Recursive types	$\mu v.\tau$	`rectype($v,`τ`)`
Bottom and top types	$\mathbf{0}, \mathbf{1}$	`F`, `T`
Logical connectives	$\wedge, \vee, \neg, \Rightarrow$	`&`, `|`, `~`, `=>`
Subtyping	$\neg(\tau_1 \leqslant \tau_2)$	`nsubtype(`τ_1`,`τ_2`,`basic_constraint`)`

The main operation `nsubtype` has 3 parameters: the two types to be compared, and the formula basic_constraint described in Section 4.2. Note that in the examples, it is not necessary for this formula to respect Def. 4.1 strictly if we know what we are doing. In particular, if only a few basic types are relevant to the example we want to try and we are not interested in the others, then we can omit these others from the formula and allow a basic constant to have no basic type at all, which we will interpret as: it has none of the *relevant* basic types. The extreme case of this is if we are not interested in basic types at all, in which case we can omit the basic_constraint parameter completely; this is interpreted as basic_constraint $= \top$.

[5]There is no guarantee that all the constraints given are *necessary* for the tree to be a counterexample, however they are always sufficient.

```
let
  $X7=((
  (let
    $X4=(CONSTANT | (PAIR & <1>$X4 & <2>$X4) | $X5),
    $X5=(FUNCTION & ((~<1>T & ~<2>T) | (<1>(PAIR & <1>$X4 &
        <2>($X4 | (~<1>T & ~<2>T & ERROR))) & <2>$X5)))
  in
    $X4) &
  (let
    $X1=(FUNCTION & (~<2>T | <2>$X1) & (~<1>T | <1>(<1>~_a | <2>(~ERROR & _g))))
  in
    $X1) &
  (let
    $X2=(FUNCTION & (~<2>T | <2>$X2) & (~<1>T | <1>(<1>~_b | <2>(~ERROR & _g))))
  in
    $X2) &
  (let
    $X6=(~FUNCTION | ((~<2>T | <2>$X6) & <2>T) |
        ((~<1>T | <1>((~<1>T | <1>(_a | _b)) & (~<2>T | <2>(ERROR | ~_g)))) & <1>T))
  in
    $X6)) | <1>$X7 | <2>$X7)
in
  $X7
```

Fig. 5. Logical translation tested for satisfiability.

Response from the solver. The solver answers either that the formula is unsatisfiable (i. e. subtyping holds) or that it has found a counterexample. This counterexample is given as a binary tree `labels (left-subtree, right-subtree)` where `labels` is a *labelling constraint* as described at the end of Sec. 6.1 above, with special symbols translated as follows: $\textcircled{\rightarrow}$ is `FUNCTION`, $\textcircled{\times}$ is `PAIR`, $\textcircled{\Omega}$ is `ERROR`, and `CONSTANT` means $\neg\textcircled{\rightarrow} \wedge \neg\textcircled{\times} \wedge \neg\textcircled{\Omega}$.

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:

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

This is formulated as follows:

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

which is automatically compiled into the logical formula shown on Figure 5 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. 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(rectype($v, (_a * _a * $v) | _NIL),
         rectype($v, (_a * $v) | _NIL))
```

which is found unsatisfiable, we prove that

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

Note that here we used a basic type, `_NIL`, but did not give a basic constraint. This means that there exist both constants which have type `_NIL` and constants which do not have it, which is what we want.

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:

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

Example (1) of the introduction allows us to illustrate the use of basic_constraint. There are two ways we can consider it: Bool could be an abbreviation for `true` $\vee$ `false` where `true` and `false` are basic types. Then basic_constraint just needs to say that `true`, `false` and `nil` are pairwise disjoint (but there also exist basic constants which belong to none of these three types). This would be written as:

```
list() = rectype($l, (_a * $l) | _NIL);
odd() = rectype($o, (_a * _a * $o) | (_a * _NIL));
even() = rectype($e, (_a * _a * $e) | _NIL);
bool() = _TRUE | _FALSE;
basic_constraint() = (_NIL => ~_TRUE & ~_FALSE) &
                     (_TRUE => ~_NIL & ~_FALSE) &
                     (_FALSE => ~_NIL & ~_TRUE);

nsubtype((odd() -> _TRUE) & (even() -> _FALSE), list() -> bool(), basic_constraint())
```

Or Bool could be a basic type, whose relation with types `true` and `false` is defined in basic_constraint:

```
list() = rectype($l, (_a * $l) | _NIL);
odd() = rectype($o, (_a * _a * $o) | (_a * _NIL));
even() = rectype($e, (_a * _a * $e) | _NIL);
basic_constraint() = (_BOOL <=> _TRUE | _FALSE) &
                     (~_NIL | ~_BOOL) &
                     (~_TRUE | ~_FALSE);

nsubtype((odd() -> _TRUE) & (even() -> _FALSE), list() -> _BOOL, basic_constraint())
```

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

Hints about non-trivial relations. Castagna and Xu [2011, section 2.7] give 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 a draft version of the work of Castagna and Xu [2011] and provided some helpful information to the authors in order to find the origin of the error and make corrections. Specifically, 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. This is formulated as follows in our system:

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

When this formula is given to the satisfiability solver the following counterexample is returned:

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

This corresponds to a domain element

$$((\llbracket\rrbracket_{\lambda_1}, \Omega_{\lambda_2}) :: \llbracket\rrbracket_{\lambda_3})_\lambda$$

such that $\alpha \notin \lambda$, $\alpha \in \lambda_1$ and $\beta \notin \lambda_3$.

$\llbracket\rrbracket$ represents the function which always diverges. This function has the property that it belongs to *any* arrow type, and it is therefore often seen in counterexamples. Indeed, it accepts any argument and never returns a result (so that it is safe to consider its result to be of any type). Here we can interpret $\llbracket\rrbracket_{\lambda_1}$ as a copy f of this function that belongs to the interpretation of α. The whole term then represents a function which is *not* in $[\![\alpha]\!]$ and which to f associates an error, while diverging on any other input.

Note that the constraint $\beta \notin \lambda_3$ is superfluous, as we can know because λ_3 is on an intermediary node, and as can be confirmed in the solver by adding `"& <2>_b"` besides the `nsubtype` statement to enforce that the right child of the root has label β.

Now, why is it a counterexample 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 counterexample function associates to it an error, which is not in $[\![\beta]\!]$.

In Section 5.1, we introduced Example (3) which we said should not hold with a sensible subtyping relation. We can check that indeed, it does not hold in our system:

```
basic_constraint() = ((_INT | _OTHER) & (~_INT | ~_OTHER));
nsubtype(_INT * _a, (_INT * ~_INT) | (_a * _INT), basic_constraint())
```

The satisfiability solver produces the following counterexample:

```
PAIR ~_INT ~_OTHER(CONSTANT _INT ~_a ~_OTHER, CONSTANT _INT _a ~_OTHER)
```

This counterexample represents a pair of two integers, where the first member of the pair is not in $[\![\alpha]\!]$ but the second member is. This shows that, with our definition, type `int` is not really indivisible: some integers have type α and others do not.

Recursive arrow types and variables. A case where recursive arrow types may naturally appear is self-application. Let us consider the function $f = \lambda x.x\ x$. The type of x must allow the function x to be applied to itself, therefore it must be an arrow type which is a subtype of its own argument type. Let α be the expected result type of our function. x must have a type τ such that $\tau \leqslant \tau \to \alpha$ holds.

One such type is the recursive type $\tau = \mu v.v \to \alpha$ (obtained by simply replacing $\leqslant$ with $=$). With this type for x, the type of f is $\tau \to \alpha$, which happens to be the same as τ (this can be checked in our solver, but is also obvious because it is exactly the unfolding of the recursion).

Another way of typing f is by using an intersection type: let β be the type of x, then x must also have type $\beta \to \alpha$, and therefore it must have type $\tau_1 = \beta \wedge (\beta \to \alpha)$. Our solver allows us to check that we have indeed $\tau_1 \leqslant \tau_1 \to \alpha$. Then f has type $\tau_2 = \tau_1 \to \alpha$. An interesting question we can ask is: how do τ_1 and τ_2 relate to τ?

The answer provided by our solver is: they are unrelated, mainly because τ does not refer to variable β. If we add this variable, we can do some interesting further comparisons: let $\tau' = \mu v.\beta \wedge (v \to \alpha)$. We then have $\tau_1 \leqslant \tau' \leqslant \tau_2$.

As a last example of the insight our solver can give on properties of types which are somehow counterintuitive, let us consider $\tau'' = \mu v.\beta \vee (v \to \alpha)$. Our solver tells us that we have $\tau' \leqslant \tau''$; but it also tells us that both τ' and τ'' are uncomparable with τ. The counterexamples it gives for $\tau' \leqslant \tau$ and $\tau \leqslant \tau''$ are the following:

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

and

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

The first one represents a function which has type β and which accepts any argument except some diverging function which does not have type β. This value does indeed have type τ', because it accepts any argument which has both type τ' and type β, and it never returns so that its result can be assumed to have type α. But it does not have type τ, since the one argument it rejects has itself type τ (the diverging function has any arrow type). However, it has type τ'', because of the β at top level, which is sufficient.

The second one represents a function which does not have type β and accepts any argument except some constant of type β. Therefore, it does not have type τ'': it rejects an argument of type β and does not have type β itself. But it does have type τ, because the only argument it rejects is not a function.

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 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 et al. [2008] provide a gentle introduction to semantic subtyping. 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 [2006] studied polymorphism in the context of regular types with arrow types. 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 of Hosoya et al. [2009]. Castagna and Xu [2011] explain 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 work of Calcagno et al. [2005] uses a spatial logic for trees as types for a lambda calculus. Their spatial logic is a fragment of the ambient logic introduced by Cardelli and Gordon [2000]. They show that validity is decidable, but the computational complexity is unknown (between PSPACE and non-elementary). No implementation is reported.

The most closely related work is the one of Castagna and Xu [2011], 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. 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 by Genevès et al. [2015]. 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 types 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 builds on the previous work of Genevès et al. [2015] since we use the satisfiability-checking algorithm defined in that article to decide the subtyping relation.

8. CONCLUSION

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

Acknowledgements. 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. 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.

Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. 2005. Deciding validity in a spatial logic for trees. *J. Funct. Program.* 15, 4 (2005), 543–572. DOI:http://dx.doi.org/10.1017/S0956796804005404

Luca Cardelli and Andrew D. Gordon. 2000. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 365–377. DOI:http://dx.doi.org/10.1145/325694.325742

Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP '11)*. ACM, Tokyo, 94–106. DOI:http://dx.doi.org/10.1145/2034773.2034788

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.

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

Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. 2015. Efficiently Deciding μ-Calculus with Converse over Finite Trees. *ACM Transactions on Computational Logic* 16, 2 (2015), 16:1–16:41. DOI:http://dx.doi.org/10.1145/2724712

Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2011. Parametric Polymorphism and Semantic Subtyping: The Logical Connection. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, Tokyo, 107–116. DOI:http://dx.doi.org/10.1145/2034773.2034789

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

Gérard P. Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (1997), 549–554.

Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. DOI:http://dx.doi.org/10.1145/197320.197383

R. Milner, L. Morris, and M. Newey. 1975. A logic for computable functions with reflexive and polymorphic types. In *Conference on Proving and Improving Programs, Arc-et-Senans, France.* IRIA-Laboria, 78150 Le Chesnay, France, 371–394.

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.

Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. 2014. XQuery 3.0: An XML Query Language, W3C Recommendation. (April 2014). http://www.w3.org/TR/xquery-30/

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