Set-theoretic Foundation of Parametric Polymorphism and Subtyping

Giuseppe Castagna[1] Zhiwu Xu[1,2] *

[1]CNRS, Laboratoire Preuves, Programmes et Systèmes, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France.
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China

Abstract. We define and study parametric polymorphism for a type system with recursive, product, union, intersection, negation, and function types. We first recall why the definition of such a system was considered hard—when not impossible—and then present the main ideas at the basis of our solution. In particular, we introduce the notion of "convexity" on which our solution is built up and discuss its connections with parametricity as defined by Reynolds to whose study our work sheds new light.

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

General Terms Theory, Languages

Keywords Types, subtyping, polymorphism, parametricity, XML

1. Introduction

The standard approach to defining subtyping is to define a collection of syntax-driven subtyping rules that relate types with similar syntactic structure. However the presence of logical operators, such as unions and intersections, makes a syntactic characterization difficult, which is why a semantic approach is used instead: type t is a subtype of type s if the set of values denoted by t is a subset of the set of values denoted by s. The goal of this study is to extend this approach to parametric types. Since such type systems are at the heart of functional languages manipulating XML data, our study directly applies to them. Parametric polymorphism has repeatedly been requested to and discussed in various working groups of standards (*eg*, RELAX NG [5] and XQuery [7]) since it would bring not only the well-known advantages already demonstrated in existing functional languages (*eg*, the typing of `map`, `fold`, and other functions that are standard in functional programming), but also new usages peculiar to XML. A typical example is SOAP [24] that provides XML "envelopes" to wrap generic content. Functions manipulating SOAP envelopes are thus working on polymorphically typed objects encapsulated in XML structures. Polymorphic higher-order functions are also needed, as shown by our practice of Ocsigen [1], a framework to develop dynamic web sites where web-site paths (uri) are associated to functions that take the uri parameters —the so-called "query strings"[8]— and return a Xhtml page. The core of the dynamic part of Ocsigen system is the function `register_new_service` whose (moral) type is:

$$\forall X{<:}\texttt{Params}.(\texttt{Path} \times (X \rightarrow \texttt{Xhtml})) \rightarrow \texttt{unit}$$

That is, it is a function that registers the association of its two parameters: a path in the site hierarchy and a function that fed with a query string that matches the description X (`Params` being the XML type of all possible query strings) returns an Xhtml page. Unfortunately, this kind of polymorphism is not available and the current implementation of `register_new_service` must bypass the type system (of OCaml, OCamlDuce [10], and/or CDuce [2]), losing all the advantages of static verification.

So why despite all this interest and motivations does no satisfactory solution exist yet? The crux of the problem is that, despite several efforts (*eg*, [17, 23]), it was not known how to define —and *a fortiori* how to decide— the subtyping relation for XML types in the presence of type variables. Actually, a purely semantic subtyping approach to polymorphism was believed to be impossible.

In this paper we focus on this problem and —though, henceforth we will seldom mention XML— solve it. The solution — which is more broadly applicable than just to an XML processing setting— is based on some strong intuitions and a lot of technical work. We follow this dichotomy for our presentation and organize it in two parts: an informal description of the main ideas and intuitions underlying the approach and the formal description of the technical development. More precisely, in Section 2 we first examine why this problem is deemed unfeasible or unpractical and simple solutions do not work (§2.1-2.3). Then we present the intuition underlying our solution (§2.4) and outline, in an informal way, the main properties that make the definition of subtyping possible (§2.5) as well as the key technical details of the algorithm that decides it (§2.6). We conclude this first part by giving some examples of the subtyping relation (§2.7) and discussing related work (§2.8). In Section 3 we present the key steps of the formal treatment to support the claims of Section 2 in particular the soundness and completeness of the algorithm and the decidability of the subtyping relation. We conclude our presentation with Section 4 where we discuss at length the connections between parametricity and our solution, as well as the new perspectives of research that our work opens. It may seem odd that we focus on a subtyping relation without defining any calculus to use it. Defining the polymorphic subtyping relation and defining a polymorphic calculus are distinct problems (though the latter depends on the former), and there is no doubt that the former is the very harder one. Once the subtyping relation is defined, it can be immediately applied to simple polymorphic calculi (*eg*, an extension with higher-order functions of the

* This author was partially supported by the National Natural Science Foundation of China under Grant No. 61070038 and by a joint training program by CNRS and the Chinese Academy of Science.

ICFP'11, September 19–21, 2011, Tokyo, Japan.

language in [17]) but the definition of calculi that fully exploit the expressiveness of these types and of algorithms that (even partially) infer type annotations are different and very difficult problems that we leave for future work.

2. The key ideas

2.1 Regular types

XML types are essentially regular tree languages: an XML type is the set of all XML documents that match the type. As such they can be encoded by product types (for concatenation), union types (for regexp unions) and recursive types (for Kleene's star). To type higher order functions we need arrow types and we also need intersection and negation types since in the presence of arrows they can no longer be encoded. Therefore, studying polymorphism for XML types is equivalent to studying it for the types that are coinductively (for recursion) produced by the following grammar:

$$t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

where b ranges over basic types (*eg*, `Bool`, `Real`, `Int`, ...), and $\mathbb{0}$ and $\mathbb{1}$ respectively denote the empty (*ie*, that contains no value) and top (*ie*, that contains all values) types. In other terms, types are nothing but a propositional logic (with standard logical connectives: $\wedge, \vee, \neg$) whose atoms are $\mathbb{0}$, $\mathbb{1}$, basic, product, and arrow types. We use $\mathcal{T}$ to denote the set of all types.

In order to preserve the semantics of XML types as sets of documents (but also to give programmers a very intuitive interpretation of types) it is advisable to interpret a type as the set of all values that have that type. Accordingly, `Int` is interpreted as the set that contains the values `0`, `1`, `-1`, `2`, ...; `Bool` is interpreted as the set the contains the values `true` and `false`; and so on. In particular, then, unions, intersections, and negations (*ie*, type *connectives*) must have a set-theoretic semantics. Formally, this corresponds to define an interpretation function from types to the sets of some domain $\mathcal{D}$ (for simplicity the reader can think of $\mathcal{D}$ as the set of all values of the language), that is $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})$, such that (*i*) $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$, (*ii*) $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$, (*iii*) $[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!]$, (*iv*) $[\![\mathbb{0}]\!] = \varnothing$, and (*v*) $[\![\mathbb{1}]\!] = \mathcal{D}$. Once we have defined such an interpretation, then the subtyping relation is naturally defined in terms of it:

$$t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$$

which, restricted to XML types, corresponds to the usual interpretation of subtyping as language containment.

2.2 Higher-order functions

All definitions above run quite smoothly as long as basic and product types are the only atoms we consider (*ie*, the setting studied by Hosoya and Pierce [18]). But as soon as we add higher-order functions, that is, arrow types, the definitions above no longer work:

1. If we take as $\mathcal{D}$ the set of all values, then this must include also λ-abstractions. Therefore, to define the semantic interpretation of types we need to define the type of λ-abstractions (in particular of the applications that may occur in their bodies) which needs the subtyping relation, which needs the semantic interpretation. We fall on a circularity.
2. If we take as $\mathcal{D}$ some mathematical domain, then we must interpret $t_1 \to t_2$ as the set of functions from $[\![t_1]\!]$ to $[\![t_2]\!]$. For instance if we consider functions as binary relations, then $[\![t_1 \to t_2]\!]$ could be the set

   $$\{\ f \subseteq \mathcal{D}^2 \mid (d_1, d_2) \in f \text{ and } d_1 \in [\![t_1]\!] \text{ implies } d_2 \in [\![t_2]\!]\ \} \quad (1)$$

   or, compactly, $\mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$, where the $\overline{S}$ denotes the complement of the set S within the appropriate universe (in words, these are the sets of pairs in which it is *not* true that the first projection belongs to $[\![t_1]\!]$ and the second does not belong to $[\![t_2]\!]$). But here the problem is not circularity but cardinality, since this would require $\mathcal{D}$ to contain $\mathscr{P}(\mathcal{D}^2)$, which is impossible.

The solution to *both* problems is given by the theory of semantic subtyping [11], and relies on the observation that in order to use types in a programming language we do not need to know what types are, but just how they are related (by subtyping). In other terms, we do not require the semantic interpretation to map arrow types into the set in (1), but just to map them into sets that induce the same subtyping relation as (1) do. Roughly speaking, this turns out to require that for all s_1, s_2, t_1, t_2, the function $[\![.]\!]$ satisfies the property:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathscr{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) \quad (2)$$

whatever the sets denoted by $s_1 \to s_2$ and $t_1 \to t_2$ are. Equation (2) above covers only the case in which we compare two single arrow types. But, of course, a similar restriction must be imposed also when comparing arbitrary Boolean combinations of arrows. Formally, this can be enforced as follows. Let $[\![.]\!]$ be a mapping from $\mathcal{T}$ to $\mathscr{P}(\mathcal{D})$, we define a new mapping $\mathbb{E}_{[\![.]\!]}$ as follows (henceforth we omit the $[\![.]\!]$ subscript from $\mathbb{E}_{[\![.]\!]}$):

$$\begin{array}{ll}
\mathbb{E}(\mathbb{0}) = \varnothing & \mathbb{E}(\mathbb{1}) = \mathcal{D} \\
\mathbb{E}(\neg t) = \mathcal{D} \setminus \mathbb{E}(t) & \mathbb{E}(b) = [\![b]\!] \\
\mathbb{E}(t_1 \vee t_2) = \mathbb{E}(t_1) \cup \mathbb{E}(t_2) & \mathbb{E}(t_1 \times t_2) = [\![t_1]\!] \times [\![t_2]\!] \\
\mathbb{E}(t_1 \wedge t_2) = \mathbb{E}(t_1) \cap \mathbb{E}(t_2) & \mathbb{E}(t_1 \to t_2) = \mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})
\end{array}$$

Then $[\![.]\!]$ and $\mathcal{D}$ form a set-theoretic *model* of types if, besides the properties (*i-v*) for the type connectives we stated in §2.1, the function $[\![.]\!]$ also satisfies the following property:

$$[\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2) \quad (3)$$

which clearly implies (2). All these definitions yield a subtyping relation with all the desired properties: type connectives (*ie*, unions, intersections, and negations) have a set-theoretic semantics, type constructors (*ie*, products and arrows) behave as set-theoretic products and function spaces, and (with some care in defining the language and its typing relation) a type can be interpreted as the set of values that have that type. All that remains to do is:

1. show that a model exists[1] (easy) and
2. show how to decide the subtyping relation (difficult).

Both points are solved in [11] and the resulting type system is at the core of the programming language ℂDuce [2].

2.3 The problem and a naive (wrong) solution

The problem we want to solve in this paper is how to extend the approach described above when we add type variables (in bold):

$$t ::= \boldsymbol{\alpha} \mid b \mid t \times t \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

where α ranges over a countable set of type variables $\mathcal{V}$. We did not include any explicit quantification for type variables: in this work (as well as, all works in the domain we are aware of, foremost [17, 23]) we focus on prenex parametric polymorphism where type quantification is meta-theoretic. Once more, the crux of the problem is how to *define* the subtyping relation between two types that contain type variables. Since we know how to subtype closed types (*ie*, types without variables), then a naive solution is to reuse this relation by considering all possible ground instances of types with variables. Let σ denote a *ground substitution*, that is

[1] We do not need to look for a particular model, since all models induce essentially the same subtyping relation: see [9] for details.

a substitution from type variables to closed types. Then according to our naive definition two types are in subtyping relation if so are their ground instances:

$$t_1 \leq t_2 \quad \stackrel{\text{def}}{\iff} \quad \forall\sigma.\ [\![t_1\sigma]\!] \subseteq [\![t_2\sigma]\!] \tag{4}$$

(provided that the domain of σ contains all the variables occurring in t_1 and t_2). This closely matches the syntactic intuition of subtyping for prenex polymorphism according to which the statement $t_1 \leq t_2$ is to be intended as $\forall\alpha_1...\alpha_n(t_1 \leq t_2)$, where $\alpha_1...\alpha_n$ are all the variables occurring in t_1 or t_2. Clearly, the containment on the right hand side of (4) is a *necessary* condition for subtyping. Unfortunately, considering it also as *necessary* and, thus, using (4) to define subtyping yields a subtyping relation that suffers too many problems to be useful.

The first obstacle is that, as conjectured by Hosoya in [17], if the subtyping relation defined by (4) is decidable (which is an open problem), then deciding it is at least as hard as the satisfiability problem for set constraint systems with *negative constraints*, which is NEXPTIME-complete and for which, so far, no practical algorithm is known.

But even if the subtyping relation defined by (4) were decidable and Hosoya's conjecture wrong, definition (4) yields a subtyping relation that misses the intuitiveness of the relation on ground types. This can be shown by an example drawn from [17]. For the sake of the example, imagine that our system includes singleton types, that is types that contain just one value, for every value of the language, and consider the following subtyping statement:

$$t \times \alpha \quad \leq \quad (t \times \neg t) \vee (\alpha \times t) \tag{5}$$

where t is a closed type.
According to (4) the statement holds if and only if $t \times s \leq (t \times \neg t) \vee (s \times t)$ holds for every closed type s. It is easy to see that the latter holds if and only if t is a singleton type. This follows from the set theoretic property that if S is a singleton, then for every set X, either $S \subseteq X$ or $X \subseteq \overline{S}$. By using this property on the singleton type t, we deduce that for every ground substitution of α either $\alpha \leq \neg t$ (therefore $t\times\alpha \leq t\times\neg t$, whence (5) follows) or $t \leq \alpha$ (therefore $t \times \alpha = (t \times \alpha\backslash t) \vee (t \times t)$ and the latter is contained component-wise in $(t \times \neg t) \vee (\alpha \times t)$, whence (5) holds again). Vice versa, if t contains at least two values, then substituting α by any singleton containing a value of t disproves the containment.

More generally, (5) holds if and only if t is an *indivisible* type, that is, a non-empty type whose only proper subtype is the empty type. Singleton types are just an example of indivisible types, but in the absence of singleton types, basic types that are pairwise disjoint are indivisible as well. Therefore, while the case of singleton types is evocative, the same problem occurs in a language with just the `Int` type, too.

Equation (5) is pivotal in our work. It gives us two reasons to think that the subtyping relation defined by (4) is unfit to be used in practice. First, it tells us that in such a system deciding subtyping is at least as difficult as deciding the indivisibility of a type. This is a very hard problem (see [3] for an instance of this problem in a simpler setting) that makes us believe more in the undecidability of the relation, than in its decidability. Second, and much worse, it completely breaks parametricity yielding a completely non-intuitive subtyping relation. Indeed notice that in the two types in (5) the type variable α occurs on the right of a product in one type and on the left of a product in the other. The idea of parametricity is that a function cannot explore arguments whose type is a type variable, it can just discard them, pass them to another function or copy them into the result. Now if (4) holds it means that by a simple subsumption a function that is parametric in its second argument can be considered parametric in its first argument instead. Understanding the intuition underlying this subtyping relation for type variables (where the same type variable may appear in unrelated positions in two related types) seems out of reach of even theoretically-oriented programmers. This is why a semantic approach for subtyping polymorphic types has been deemed unfeasible and discarded in favor of partial or syntactic solutions (see related works in §2.8).

2.4 Ideas for a solution

Although the problems we pointed out in [17] are substantial, they do not preclude a semantic approach to parametric polymorphism. Furthermore the shortcomings caused by the absence of this approach make the study well worth of trying. Here we show that—paraphrasing a famous article by John Reynolds [22]—subtyping of polymorphism *is* set-theoretic.

The conjecture that we have been following since we discovered the problem of [17], and that is at the basis of all this work, is that *the loss of parametricity is only due to the behavior of indivisible types*, all the rest works (more or less) smoothly. The crux of the problem is that for an indivisible type t the validity of the formula

$$t \leq \alpha \quad \text{or} \quad \alpha \leq \neg t \tag{6}$$

can *stutter* from one subformula to the other (according to the assignment of α) losing in this way the uniformity typical of parametricity. If we can give a *semantic* characterization of models in which *stuttering* is absent, we believed this would have yielded a subtyping relation that is (i) semantic, (ii) intuitive for the programmer,[2] and (iii) decidable. The problem with indivisible types is that they are either completely inside or completely outside any other type. What we need, then, is to make indivisible types "splittable", so that type variables can range over strict subsets of any type, indivisible ones included. Since this is impossible at a syntactic level, we shall do it at a semantic level. First, we replace ground substitutions with semantic (set) assignments of type variables, $\eta : \mathcal{V} \to \mathscr{P}(\mathcal{D})$, and add to interpretation functions a semantic assignment as a further parameter (as is customary in denotational semantics):

$$[\![.]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$$

Such an interpretation (actually, the pair $([\![.]\!], \mathcal{D})$) is then a *set-theoretic model* if and only if for all assignments η it satisfies the following conditions (in bold the condition that shows the role of the assignment parameter η):

$$\begin{array}{ll} \boldsymbol{[\![\alpha]\!]\eta = \eta(\alpha)} & [\![\neg t]\!]\eta = \mathcal{D}\backslash[\![t]\!]\eta \\ [\![\mathbb{0}]\!]\eta = \varnothing & [\![t_1 \vee t_2]\!]\eta = [\![t_1]\!]\eta \cup [\![t_2]\!]\eta \\ [\![\mathbb{1}]\!]\eta = \mathcal{D} & [\![t_1 \wedge t_2]\!]\eta = [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \end{array}$$
$$[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \mathbb{E}(t_1)\eta \subseteq \mathbb{E}(t_2)\eta$$

(where $\mathbb{E}()$ is extended in the obvious way to cope with semantic assignments). Then the subtyping relation is defined as follows:

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \forall\eta\in\mathscr{P}(\mathcal{D})^{\mathcal{V}}.\ [\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \tag{7}$$

In this setting, every type t that denotes a set of at least two elements of $\mathcal{D}$ can be split by an assignment. That is, it is possible to define an assignment for which a type variable α denotes a subset of $\mathcal{D}$ that is neither completely inside nor completely outside the interpretation of t. Therefore for such a type t, neither equation (6) nor, *a fortiori*, equation (5) hold. It is then clear that the stuttering of (6) is absent in every set-theoretic model in which all non-empty types—indivisible types included—denote *infinite* subsets of $\mathcal{D}$. Infinite denotations for non-empty types look as a possible, though

[2] For instance, type variables can only be subsumed to themselves and according to whether they occur in a covariant or contravariant position, to $\mathbb{1}$ and to unions in which they explicitly appear or to $\mathbb{0}$ and intersections in which they explicitly appear, respectively. De Morgan's laws can be used to reduce other cases to one of these.

specific, solution to the problem of indivisible types. But what we are looking for is not a particular solution. We are looking for a semantic characterization of the "uniformity" that characterizes parametricity, in order to define a subtyping relation that is, we repeat, semantic, intuitive, and decidable.

This characterization is provided by the property of *convexity*.

2.5 Convexity

A set theoretic model $([\![.]\!], \mathcal{D})$ is *convex* if and only if for every finite set of types $t_1,\ldots, t_n$ it satisfies the following property:

$$\begin{array}{c}\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } \cdots \text{ or } [\![t_n]\!]\eta{=}\varnothing)\\ \Longleftrightarrow \\ (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.[\![t_n]\!]\eta{=}\varnothing)\end{array} \tag{8}$$

This property is the cornerstone of our approach. As such it deserves detailed comments. It states that, given any finite set of types, if every assignment makes some of these types empty, then it is so because there exists one particular type that is empty for all possible assignments.[3] Therefore convexity forces the interpretation function to behave uniformly on its zeros (*ie*, on types whose interpretation is the empty set). Now, the zeros of the interpretation function play a crucial role in the theory of semantic subtyping, since they completely characterize the subtyping relation. Indeed $s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} \subseteq \varnothing \Leftrightarrow [\![s \wedge \neg t]\!] = \varnothing$. Consequently, checking whether $s \leq t$ is equivalent to checking whether the type $s \wedge \neg t$ is empty; likewise, equation (3) in §2.2 is equivalent to requiring that for all t it satisfies $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$. We deduce that convexity forces the subtyping relation to have a uniform behavior and, ergo, rules out non-intuitive relations such as the one in (5). This is so because convexity prevents stuttering, insofar as in every convex model ($[\![t{\wedge}\neg\alpha]\!]\eta{=}\varnothing$ or $[\![t{\wedge}\alpha]\!]\eta{=}\varnothing$) holds for all assignments η if and only if t is empty.

Convexity is the property we seek. The resulting subtyping relation is semantically defined and preserves the set-theoretic semantics of type connectives (union, intersection, negation) and the containment behavior of set-theoretic interpretations of type constructors (set-theoretic products for product types and set-theoretic function spaces for arrow types). Furthermore, the subtyping relation is not only semantic but also intuitive. First, it excludes non-intuitive relations by imposing a uniform behavior distinctive of the parametricity *à la* Reynolds: as we discuss at length in the conclusion, we push the analogy much farther since we believe that parametricity and convexity are connected, despite the fact that the former is defined in terms of *transformations* of related terms while the latter deals only with (subtyping) relations. Second, it is very easy to explain the intuition of type variables to a programmer:

> For what concerns subtyping, a type variable can be considered as a special new user-defined basic type that is unrelated to any other atom but $\mathbb{0}$ and $\mathbb{1}$ and itself.[4] Type variables are special because their intersection with any ground type may be non-empty, whatever this type is.

[3] We dubbed this property *convexity* after convex formulas: a formula is convex if whenever it entails a disjunction of formulas, then it entails one of them. The $\Rightarrow$ direction of (8) (the other direction is trivial) states the convexity of assignments with respect to emptiness: $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \Rightarrow \bigvee_{i\in I}[\![t_i]\!]\eta = \varnothing$ implies that there exists $h \in I$ such that $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \Rightarrow [\![t_h]\!]\eta = \varnothing$.

[4] This holds true even for languages with bounded quantification which, as it is well known, defines the subtyping relation for type variables. Bounded quantification does not require any modification to our system, since it can be encoded by intersections: a type variable α bounded by a type t can be encoded by a fresh (unbounded) variable β by replacing $\beta{\wedge}t$ for every occurrence of α. We can do even more, since by using intersections we can impose different bounds to different occurrences of the same variable.

Of course, neither in the theoretical development nor in the subtyping algorithm type variables are dealt with as basic types. They need very subtle and elaborated techniques that form the core of our work. But this complexity is completely transparent to the programmer which can thus rely on a very simple intuition.

All that remains to do is (i) to prove the convexity property is not too restrictive, that is, that there exists at least one convex set-theoretic model and (ii) to show an algorithm that checks the subtyping relation. Contrary to the ground case, *both* problems are difficult. While their solutions require a lot of technical results (see Section 3), the intuition underlying them is relatively simple. For what concerns the existence of a convex set-theoretic model, the intuition can be grasped by considering just the logical fragment of our types, that is, the types in which $\mathbb{0}$ and $\mathbb{1}$ are the only atoms. This corresponds to the (classical) propositional logic where the two atoms represent, respectively, `false` and `true`. Next, consider the instance of the convexity property given for just two types, t_1 and t_2. It is possible to prove that every non-degenerate Boolean algebra (*ie*, every Boolean algebra with more than two elements) satisfies it. Reasoning by induction it is possible to prove that convexity for n types is satisfied by any Boolean algebra containing at least $n + 1!$ elements and from there deduce that all infinite Boolean algebras satisfy convexity. It is then possible to extend the proof to the case that includes basic, product, and arrow types and deduce the following result:

> Every set-theoretic model of closed types in which non-empty types denote infinite sets is a convex set-theoretic model for the polymorphic types.

Therefore, not only do we have a large class of convex models, but also we recover our initial intuition that models with infinite denotations was the way to go.

All that remains to explain is the subtype checking algorithm. We do it in the next section, but before we want to address the possible doubts of a reader about what the denotation of a "finite" type like `Bool` is in such models. In particular, since this denotation contains not only (the denotations of) `true` and `false` but infinitely many other elements, then the reader can rightly wonder what these other elements are and whether they carry any intuitive meaning. In order to explain this point, let us first reexamine what convexity does for infinite types. Convexity is a condition that makes the interpretation of subtyping in some sense independent from the particular syntax of types. Imagine that the syntax of types includes just one basic type: `Int`. Then `Int` is an indivisible type and therefore there exist non-convex models in which the following relation (which is an instance of equation (5) of Section 2.3) holds.

$$\texttt{Int} \times \alpha \quad \leq \quad (\texttt{Int} \times \neg\texttt{Int}) \vee (\alpha \times \texttt{Int}) \tag{9}$$

(*eg*, a model where `Int` is interpreted by a singleton set: in a non-convex model nothing prevents such an interpretation). Now imagine to add the type `Odd`, subtype of `Int`, to the type system: then (9) no longer holds (precisely, the interpretation at issue no longer is a model) since the substitution of α by `Odd` disproves it. Should the presence of `Odd` change the containment relation between `Int` and the other types? Semantically this should not happen. A relation as (9) should have the same meaning independently from whether `Odd` is included in the syntax of types or not. In other terms we want the addition of `Odd` to yield a conservative extension of the subtyping relation. Therefore, all models in which (9) is valid must be discarded. Convexity does it.

The point is that convexity pushes this idea to all types, so that their interpretation is independent from the possible syntactic subtypes they may have. It is as if the interpretation of subtyping assumed that every type has at least one (actually, infinitely many) stricter non empty subtype(s). So what could the denotation of type

Bool be in such a model, then? A possible choice is to interpret Bool into a set containing labeled versions of true and false, where labels are drawn from an infinite set of labels (a similar interpretation was first introduced by Gesbert *et al.* [15]: see Section 2.8 on related work). Here the singleton type {true} is interpreted as an infinite set containing differently labeled versions of the single element true. Does this labeling carry any intuitive meaning? One can think of it as representing name subtyping: these labels are the names of subtypes of the singleton type {true} for which the subtyping relation is defined by name subtyping. As we do not want the subtyping relation for Int to change (non conservatively) when adding to the system the type Odd, so for the same reason we do not want the subtyping relation for singleton types to change when adding by name subtyping new subtypes, even when these subtypes are subtypes of a singleton type. So convexity makes the subtyping relation insensitive to possible extensions by name subtyping.

2.6 Subtyping algorithm

The subtyping algorithm for the relation induced by convex models can be decomposed in 6 elementary steps. Let us explain the intuition underlying each of them: all missing details can be found in Section 3.

First of all, we already said that deciding $t_1 \leq t_2$—*ie*, whether for all η, $[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta$—is equivalent to decide the emptiness of the type $t_1 \wedge \neg t_2$—*ie*, whether for all η, $[\![t_1 \wedge \neg t_2]\!]\eta = \varnothing$—. So the first step of the algorithm is to transform the problem $t_1 \leq t_2$ into the problem $t_1 \wedge \neg t_2 \leq \mathbb{0}$:

Step 1: *transform the subtyping problem into an emptiness decision problem.*

Our types are just a propositional logic whose atoms are type variables, $\mathbb{0}$, $\mathbb{1}$, basic, product, and arrow types. We use a to range over atoms and, following the logic nomenclature, call *literal*, ranged over by ℓ, an atom or its negation:

$$a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha \qquad \ell ::= a \mid \neg a$$

By using the laws of propositional logic we can transform every type into a disjunctive normal form, that is, into a union of intersections of literals:

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

Since the interpretation function preserves the set-theoretic semantics of type connectives, then every type is empty if and only if its disjunctive normal form is empty. So the second step of our algorithm consists of transforming the type $t_1 \wedge \neg t_2$ whose emptiness was to be checked, into a disjunctive normal form:

Step 2: *put the type whose emptiness is to be decided in a disjunctive normal form.*

Next, we have to decide when a normal form, that is, a union of intersections, is empty. A union is empty if and only if every member of the union is empty. Therefore the problem reduces to deciding emptiness of an intersection of literals: $\bigwedge_{i \in I} \ell_i$. Intersections of literals can be straightforwardly simplified. Every occurrence of the literal $\mathbb{1}$ can be erased since it does not change the result of the intersection. If either any of the literals is $\mathbb{0}$ or two literals are a variable and its negation, then we do not have to perform further checks since the intersection is surely empty. An intersection can be simplified also when two literals with different constructors occur in it: if in the intersections there are two atoms of different constructors, say, $t_1 \times t_2$ and $t_1 \to t_2$, then their intersection is empty and so is the whole intersection; if one of the two atoms is negated, say, $t_1 \times t_2$ and $\neg(t_1 \to t_2)$, then it can be eliminated since it contains the one that is not negated; if both atoms are negated, then the intersection can also be simplified (with some more work: *cf.* the formal development in Section 3). Therefore the third step of the algorithm is to perform these simplifications so that the problem is reduced to deciding emptiness of intersections that are formed by literals that are (possible negations of) either type variables or atoms all of the same constructor (all basic, all product, or all arrow types):

Step 3: *simplify mixed intersections.*

At this stage we have to decide emptiness of intersections of the form $\bigwedge_{i \in I} a_i \wedge \bigwedge_{j \in J} \neg a'_j \wedge \bigwedge_{h \in H} \alpha_h \wedge \bigwedge_{k \in K} \neg \beta_k$ where all the a_i's and a'_j's are atoms with the same constructor, and where $\{\alpha_h\}_{h \in H}$ and $\{\beta_k\}_{k \in K}$ are disjoint sets of type variables: we just reordered literals so that negated variables and the other negated atoms are grouped together. In this step we want to get rid of the rightmost group in the intersection, that is, the one with negated type variables. In other terms, we want to reduce our problem to deciding the emptiness of an intersections as the above, but where all top-level occurrences of type variables are positive. This is quite easy, and stems from the observation that if a type with a type variable α is empty for every possible assignment of α, then it will be empty also if one replaces $\neg\alpha$ for α in it: exactly the same checks will be performed since the denotation of the first type for $\alpha \mapsto S \subseteq \mathcal{D}$ will be equal to the denotation of the second type for $\alpha \mapsto \overline{S} \subseteq \mathcal{D}$. That is to say, $\forall \eta. [\![t]\!]\eta = \varnothing$ if and only if $\forall \eta. [\![t\{^{\neg\alpha}/_{\alpha}\}]\!]\eta = \varnothing$ (where $t\{^{t'}/_{\alpha}\}$ denotes the substitution in t of t' for α). So all the negations of the group of toplevel negated variables can be eliminated by substituting $\neg\beta_k$ for β_k in the a_i's and a'_j's:

Step 4: *eliminate toplevel negative variables.*

Next comes what probably is the trickiest step of the algorithm. We have to prove emptiness of intersections of atoms a_i and negated atoms a'_j all on the same constructors and of positive variables α_k. To lighten the presentation let us consider just the case in which atoms are all product types (the case for arrow types is similar though trickier, while the case for basic types is trivial since it reduces to the case for basic types without variables). By using De Morgan's laws we can move negated atoms on the right hand-side of the relation so that we have to check the following containment

$$\bigwedge_{t_1 \times t_2 \in P} t_1 \times t_2 \wedge \bigwedge_{h \in H} \alpha_h \quad \leq \quad \bigvee_{t'_1 \times t'_2 \in N} t'_1 \times t'_2 \qquad (10)$$

where P and N respectively denote the sets of positive and negative atoms. Our goal is to eliminate all top-level occurrences of variables (the α_h's) so that the problem is reduced to checking emptiness of product literals. To that end observe that each α_h is intersected with other products. Therefore whatever the interpretation of α_h is, the only part of its denotation that matters is the one that intersects $\mathcal{D}^2$. Ergo, it is useless, at least at top-level, to check all possible assignments for α_h, since those contained in $\mathcal{D}^2$ will suffice. These can be checked by replacing $\gamma_h^1 \times \gamma_h^2$ for α_h, where γ_h^1, γ_h^2 are fresh type variables. Of course the above reasoning holds for the top-level variables, but nothing tells us that the non top-level occurrences of α_h will intersect any product. So replacing them with just $\gamma_h^1 \times \gamma_h^2$ would yield a sound but incomplete check. We rather replace every non toplevel occurrence of α_h by $(\gamma_h^1 \times \gamma_h^2) \vee \alpha_h$. This still is a sound substitution since if (10) holds, then it must also hold for the case where $(\gamma_h^1 \times \gamma_h^2) \vee \alpha_h$ is substituted for α_h (with the $\vee\alpha_h$ part useless for toplevel occurrences). Rather surprisingly, at least at first sight, this substitution is also complete, that is (10) holds if and only if the following holds:

$$\bigwedge_{t_1 \times t_2 \in P} t_1\theta \times t_2\theta \wedge \bigwedge_{h \in H} \gamma_h^1 \times \gamma_h^2 \quad \leq \quad \bigvee_{t'_1 \times t'_2 \in N} t'_1\theta \times t'_2\theta$$

where θ is the substitution $\{(\gamma_h^1\times\gamma_h^2)\vee\alpha_h/_{\alpha_h}\}_{h\in H}$.[5] As an aside, we signal that this transformation holds only because α_h's are positive: the application of *Step 4* is thus a necessary precondition to the application of this one. We thus succeeded to eliminate all toplevel occurrences of type variables and, thus, we reduced the initial problem to the problem of deciding emptiness of intersections in which all literals are products or negations of products (and similarly for arrows):

Step 5: *eliminate toplevel variables.*

The final step of our algorithm must decompose the type constructors occurring at toplevel in order to recurse or stop. To that end it will use set-theoretic properties to deconstruct atom types and, above all, the convexity property to decompose the emptiness problem into a set of emptiness subproblems (this is where convexity plays an irreplaceable role: without convexity the definition of an algorithm seems to be out of our reach). Let us continue with our example with products. At this stage all it remains to solve is to decide a containment of the following form (we included the products of fresh variables into P):

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2 \tag{11}$$

Using the set-theoretic properties of the interpretation function and our definition of subtyping, we can prove (see Lemma 6.4 in [11] for details) that (11) holds if and only if for all $N'\subseteq N$,

$$\forall\eta.\left(\llbracket \bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t'_1\times t'_2\in N'}\neg t'_1\rrbracket\eta=\varnothing \ \text{ or }\ \llbracket \bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t'_1\times t'_2\in N\setminus N'}\neg t'_2\rrbracket\eta=\varnothing\right)$$

We can now apply the convexity property and distribute the quantification on η on each subformula of the or. This is equivalent to state that we have to check the emptiness for each type that occurs as argument of the interpretation function. Playing a little more with De Morgan's laws and applying the definition of subtyping we can thus prove that (11) holds if and only if

$$\forall N'\subseteq N.\left(\bigwedge_{t_1\times t_2\in P} t_1 \leq \bigvee_{t'_1\times t'_2\in N'} t'_1\right) \text{ or } \left(\bigwedge_{t_1\times t_2\in P} t_2 \leq \bigvee_{t'_1\times t'_2\in N\setminus N'} t'_2\right)$$

To understand the rationale of this transformation the reader can consider the case in which both P and N contain just one atom, namely, the case for $t_1\times t_2 \leq t'_1\times t'_2$. There are just two cases to check ($N'=\varnothing$ and $N'=N$) and it is not difficult to see that the condition above becomes: $(t_1\leq\mathbb{0})$ or $(t_2\leq\mathbb{0})$ or $(t_1\leq t'_1$ and $t_2\leq t'_2)$, as expected.

The important point however is that we were able to express the problem of (11) in terms of subproblems that rest on strict subterms (there is a similar decomposition rule for arrow types). Remember that our types are possibly infinite trees since they were *coinductively* generated by the grammar in §2.1. We do not consider every possible coinductively generated tree, but only those that are regular (*ie*, that have a finite number of distinct subtrees) and in which every infinite branch contains infinitely many occurrences of type constructors (ie, products and arrows). The last condition rules out meaningless terms (such as $t=\neg t$) as well as infinite unions and intersections. It also provides a well-founded order that allows us to use recursion. Therefore, we memoize the relation in (11) and recursively call the algorithm from *Step 1* on the subterms we obtained from decomposing the toplevel constructors:

Step 6: *eliminate toplevel constructors, memoize, and recurse.*

The algorithm is sound and complete with respect to the subtyping relation defined by (7) and terminates on all types (which implies the decidability of the subtyping relation).

2.7 Examples

The purpose of this subsection is twofold: first, we want to give some examples to convey the idea that the subtyping relation is intuitive; second we present some cases that justify the subtler and more technical aspects of the subtyping algorithm we exposed in the previous subsection. All the examples below can be tested in our prototype subtype-checker.

In what follows we will use x, y, z to range over recursion variables and the notation $\mu x.t$ to denote recursive types. This should suffice to avoid confusion with free type variables that are ranged over by α, β, and γ.

As a first example we show how to use type variables to internalize meta-properties. For instance, for all ground types t_1, t_2, and t_3 the relation $(t_1\to t_3)\wedge(t_2\to t_3)\leq(t_1\vee t_2)\to t_3$ and its converse hold. This meta-theoretic property can be expressed in our type system since the following relation holds:

$$(\alpha\to\gamma)\wedge(\beta\to\gamma)\ \sim\ (\alpha\vee\beta)\to\gamma$$

(where $\sim$ denotes that both $\leq$ and $\geq$ hold). Of course we can apply this generalization to any relation that holds for generic types. For instance, we can prove common distributive laws such as

$$((\alpha\vee\beta)\times\gamma)\ \sim\ (\alpha\times\gamma)\vee(\beta\times\gamma) \tag{12}$$

and combine it with the previous relation and the covariance of arrow on codomains to deduce

$$(\alpha\times\gamma\to\delta_1)\wedge(\beta\times\gamma\to\delta_2)\ \leq\ ((\alpha\vee\beta)\times\gamma)\to\delta_1\vee\delta_2$$

Similarly we can prove that the set of lists whose elements have type α, that is, $\alpha\ \mathsf{list}=\mu x.(\alpha\times x)\vee\mathsf{nil}$, contains both the α-lists with an even number of elements

$$\mu x.(\alpha\times(\alpha\times x))\vee\mathsf{nil}\ \leq\ \mu x.(\alpha\times x)\vee\mathsf{nil}$$

(where nil denotes the singleton type containing just the value nil) and the α-lists with an odd number of elements

$$\mu x.(\alpha\times(\alpha\times x))\vee(\alpha\times\mathsf{nil})\ \leq\ \mu x.(\alpha\times x)\vee\mathsf{nil}$$

and it is itself contained in the union of the two, that is:

$$\alpha\ \mathsf{list}\ \sim\ (\mu x.(\alpha\times(\alpha\times x))\vee\mathsf{nil})\vee(\mu x.(\alpha\times(\alpha\times x))\vee(\alpha\times\mathsf{nil}))$$

We said that the intuition for subtyping type variables is to consider them as basic types. But type variables are *not* basic types. As an example, if t is a non-empty type, then we have that:

$$\alpha\wedge(\alpha\times t)\ \not\leq\ t_1\to t_2$$

which implies that $\alpha\wedge(\alpha\times t)$ is not empty. This is correct because if for instance we substitute the type $t\vee(t\times t)$ for α, then (by the distributivity law stated in (12)) the intersection is equal to $(t\times t)$, which is non-empty. However, note that if α were a basic type, then the intersection $\alpha\wedge(\alpha\times t)$ would be empty. Furthermore, since the following relation holds

$$\alpha\wedge(\alpha\times t)\ \leq\ \alpha$$

then, this last containment is an example of non-trivial containment (in the sense that the left hand-side is not empty) involving type variables. For an example of non-trivial containment involving arrows the reader can check

$$\mathbb{1}\to\mathbb{0}\ \leq\ \alpha\to\beta\ \leq\mathbb{0}\to\mathbb{1}$$

which states that $\mathbb{1}\to\mathbb{0}$, the set of all functions that diverge on all arguments, is contained in all arrow types $\alpha\to\beta$ (whatever types

[5] Note that the result of this substitution is equivalent to using the substitution $\{(\gamma_h^1\times\gamma_h^2)\vee\gamma_h^3/_{\alpha_h}\}_{h\in H}$ where γ_h^3 is also a fresh variable: we just spare a new variable by reusing α_h which would be no longer used (actually this artifice makes proofs much easier).

α and β are) and that the latter are contained in $\mathbb{0} \to \mathbb{1}$, which is the set of all function values.

Type connectives implement classic proposition logic. If we use $\alpha \Rightarrow \beta$ to denote $\neg\alpha \vee \beta$, that is logical implication, then the following subtyping relation is a proof of Pierce's law:

$$\mathbb{1} \;\leq\; ((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha$$

since being a supertype of $\mathbb{1}$ logically corresponds to being equivalent to true (note that arrow types do not represent logical implication; for instance, $\mathbb{0}\to\mathbb{1}$ is not empty: it contains all function values). Similarly, the system captures the fundamental property that for all non-empty sets β the set $(\beta\wedge\alpha) \vee (\beta\wedge\neg\alpha)$ is never empty:

$$(\beta \wedge \alpha) \vee (\beta \wedge \neg\alpha) \;\sim\; \beta$$

from which we can derive

$$\mathbb{1} \;\leq\; (\,((\beta\wedge\alpha) \vee (\beta\wedge\neg\alpha)) \Rightarrow \mathbb{0}\,) \;\Rightarrow\; (\beta \Rightarrow \mathbb{0})$$

This last relation can be read as follows: if $(\beta\wedge\alpha) \vee (\beta\wedge\neg\alpha)$ is empty, then β is empty.

But the property above will never show a stuttering validity since the algorithm returns false when asked to prove

$$\mathsf{nil} \times \alpha \quad\leq\quad (\mathsf{nil} \times \neg\mathsf{nil}) \vee (\alpha \times \mathsf{nil})$$

even for a singleton type as nil.

The subtyping relation has some simple form of introspection since $t_1 \leq t_2$ if and only if $\mathbb{1} \leq t_1 \Rightarrow t_2$ (*ie*, by negating both types and reversing the subtyping relation, $t_1 \wedge \neg t_2 \leq \mathbb{0}$). However, the introspection capability is very limited insofar as it is possible to state interesting properties only when atoms are type variables: although we can characterize the subtyping relation $\leq$, we have no idea about how to characterize its negation $\not\leq$. [6]

The necessity for the tricky substitution $\alpha \mapsto (\gamma_1 \times \gamma_2) \vee \alpha$ performed at *Step 5* of the algorithm can be understood by considering the following example where t is any non-empty type:

$$(\alpha \times t) \wedge \alpha \;\leq\; ((\mathbb{1} \times \mathbb{1}) \times t).$$

If in order to check the relation above we substituted just $\gamma_1 \times \gamma_2$ for α, then this would yield a positive result, which is wrong: if we replace α by $b \vee (b \times t)$, where b is any basic type, then the intersection on the left becomes $(b \times t)$ and b is neither contained in $\mathbb{1} \times \mathbb{1}$ nor empty. Our algorithm correctly disproves the containment, since it checks also the substitution of $(\gamma_1 \times \gamma_2) \vee \alpha$ for the first occurrence of α, which captures the above counterexample.

Finally, the system also proves subtler relations whose meaning is not clear at first sight, such as:

$$\alpha_1 \to \beta_1 \;\leq\; ((\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)) \vee \neg(\alpha_2 \to (\beta_2 \wedge \neg\beta_1)) \quad (13)$$

In order to prove it, the subtyping algorithm first moves the occurrence of $\alpha_2 \to (\beta_2 \wedge \neg\beta_1)$ from the right of the subtyping relation to its left: $(\alpha_1 \to \beta_1) \wedge (\alpha_2 \to (\beta_2 \wedge \neg\beta_1)) \leq ((\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2))$; then following the decomposition rules for arrows the algorithm checks the four following cases (Step 6 of the algorithm), which hold straightforwardly:

$$\begin{cases} \alpha_1 \wedge \alpha_2 \leq \mathbb{0} \text{ or } \beta_1 \wedge (\beta_2 \wedge \neg\beta_1) \leq \beta_1 \wedge \beta_2 \\ \alpha_1 \wedge \alpha_2 \leq \alpha_1 \text{ or } (\beta_2 \wedge \neg\beta_1) \leq \beta_1 \wedge \beta_2 \\ \alpha_1 \wedge \alpha_2 \leq \alpha_2 \text{ or } \beta_1 \leq \beta_1 \wedge \beta_2 \\ \alpha_1 \wedge \alpha_2 \leq \alpha_1 \vee \alpha_2 \end{cases}$$

Notice that relation (13) is quite subtle insofar as neither $\alpha_1 \to \beta_1 \leq (\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)$ nor $\alpha_1 \to \beta_1 \leq \neg(\alpha_2 \to (\beta_2 \wedge \neg\beta_1))$ hold: the type on left hand-side of (13) is contained in the union of the two types on the right hand-side of (13) without being completely contained in either of them.

2.8 Related work

This work extends the work on semantic subtyping [11], as such the two works share the same approach and common developments. Since convex models of our theory can be derived from the models of [11], then several techniques we used in our subtyping algorithm (in particular the decomposition of toplevel type constructors) are directly issued from the research in [11]. This work starts precisely from where [11] stopped, that is the monomorphic case, and adds prenex parametric polymorphism to it.

The most advanced work on polymorphism for XML types, thus far, is the Hosoya, Frisch, and Castagna's approach described in [17], whose extended abstract was first presented at POPL '05. Together with [11], the paper by Hosoya, Frisch, and Castagna constitutes the starting point of this work. A starting point that, so far, was rather considered to establish a (negative) final point. As a matter of fact, although the polymorphic system in [17] is the one used to define the polymorphic extension of XDuce [18] (incorporated from version 0.5.0 of the language), the three authors of [17] agree that the main interest of their work does not reside in its type system, but rather in the negative results that motivate it. In particular, the pivotal example of our work, equation (5), was first presented in [17], and used there to corroborate the idea that a purely semantic approach for polymorphism of regular tree types was an hopeless quest. At that time, this seemed so more hopeless that the equation (5) did not involve arrow types: a semantically defined polymorphic subtyping looked out of reach even in the restrictive setting of Hosoya and Pierce seminal work [18], which did not account for higher-order functions. This is why [17] falls back on a syntactic approach that, even if it retains some flavors of semantic subtyping, it cannot be extended to higher-order functions (a lack that nevertheless fits XDuce). Our works shows that the negative results of [17] were not so insurmountable as it had been thought.

Hitherto, the only work that blends polymorphic regular types and arrow types is Jérôme Vouillon's work that was presented at POPL '06 [23]. His approach, however, is very different from ours insofar as it is intrinsically syntactic. Vouillon starts from a particular language (actually, a peculiar pattern algebra) and coinductively builds up on it the subtyping relation by a set of inference rules. The type algebra includes only the union connective (negation and intersection are missing) and a semantic interpretation of subtyping is given *a posteriori* by showing that a pattern (types are special cases of patterns) can be considered as the set of values that match the pattern. Nevertheless, this interpretation is still syntactic in nature since it relies on the definition of matching and containment, yielding a system tailored for the peculiar language of the paper. This allows Vouillon to state impressive and elegant results such as the translation of the calculus into a non-explicitly-typed one, or the interpretation of open types containment as in our equation (4) (according to Vouillon this last result is made possible in his system by the absence of intersection types, although the critical example in (5) does not involve any intersection). But the price to pay is a system that lacks naturalness (*eg*, the wild-card pattern has different meanings according to whether it occurs in the right or in left type of a subtyping relation) and, even more, it lacks the generality of our approach (we did not state our subtype system for any specific language while Vouillon's system is inherently tied to a particular language whose semantics it completely relies on). The semantics

[6] For instance, it would be nice to prove something like:

$$(\neg\beta_1 \vee ((\beta_1 \Rightarrow \alpha_1) \wedge (\alpha_2 \Rightarrow \beta_2))) \;\sim\; (\alpha_1 \to \alpha_2 \Rightarrow \beta_1 \to \beta_2)$$

since it seems to provide a complete characterization of the subtyping relation between two arrow types. Unfortunately the equivalence is false since $\beta_1 \not\leq \alpha_1$ does not imply $\beta_1 \wedge \neg\alpha_1 \geq \mathbb{1}$ but just $\beta_1 \wedge \neg\alpha_1 \not\leq \mathbb{0}$. This property can be stated only at meta level, that is: $\alpha_1 \to \alpha_2 \leq \beta_1 \to \beta_2$ if and only if $(\beta_1 \leq \mathbb{0}$ or $(\beta_1 \leq \alpha_1$ and $\alpha_2 \leq \beta_2))$.

of Vouillon's patterns is so different from ours that typing Vouillon's language with our types seems quite difficult.

Other works less related to ours are those in which XML and polymorphism are loosely coupled. This is the case of OCamlDuce [10] where ML-polymorphism and XML types and patterns are merged together without mixing: the main limitation of this approach is that it does not allow parametric polymorphism for XML types, which is the whole point of our (and Vouillon's) work(s). A similar remark can be done for Xtatic [12] that merges C# name subtyping with the XDuce set-theoretic subtyping and for XHaskell [19] whose main focus is to implement XML subtyping using Haskell's type-classes. A more thorough comparison of these approaches can be found in [10, 17].

Polymorphism can be attained by adopting the so-called data-binding approach which consists in encoding XML types and values into the structures of an existing polymorphic programming language. This is the approach followed by HaXML [26]. While the polymorphism is inherited from the target language, the rigid encoding of XML data into fixed structures loses all flexibility of the XML type equivalences so as, for instance, $(t{\times}s_1) \vee (t{\times}s_2)$ and $(t{\times}s_1{\vee}s_2)$ are different (and even unrelated) types.

Our work already has a follow-up. In a paper included in these same proceedings [15] Gesbert, Genevès, and Layaïda use the framework we define here to give a different decision procedure for our subtyping relation. More precisely, they take a specific model for the monomorphic type system (*ie*, the model defined by Frisch *et al.* [11] and used by the language ℂDuce), they encode the subtyping relation induced by this model into a tree logic, and use a satisfiability solver to efficiently decide it. Next, they extend the type system with type variables and they obtain a convex model by interpreting non-empty types as infinite sets using a labeling technique similar to the one we outlined at the end of Section 2.5: they label values by (finite sets of) type variables and every non empty ground type is, thus, interpreted as an infinite set containing the infinitely many labelings of its values. Again the satisfiability solver provides a decision procedure for the subtyping relation. Their technique is interesting in several respects. First it provides a very elegant solution to the problem of deciding our subtyping relation, solution that is completely different from the one given here. Second, their technique shows that the decision problem is EXPTIME, (while here we only prove the decidability of the problem by showing the termination of our algorithm). Finally, their logical encoding paves the way to extending types (and subtyping) with more expressive logical constraints representable by their tree logic. In contrast, our algorithm is interesting for quite different reasons: first, it is defined for generic interpretations rather than for a fixed model; second, it shows how convexity is used in practice (see in particular ***Step 6*** of the algorithm); and, finally, our algorithm is a straightforward modification of the algorithm used in ℂDuce and, as such, can benefit of the technology and optimizations used there.[7] We expect the integration of this subtyping relation in the ℂDuce to be available in the near future.

Finally, we signal the work on polymorphic iterators for XML presented in [4]. It introduces a very simple strongly normalizing calculus fashioned to define tree iterators. These iterators are lightly checked at the moment of their definition: the compiler does not complain unless they are irremediably flawed. This optimistic typing, combined with the relatively limited expressive power of the calculus, makes it possible to type iterator applications in a very precise way (essentially, by performing an abstract execution of the iterator on the types) yielding a kind of polymorphism that is out of reach of parametric or subtype polymorphism (for instance it can precisely type the reverse function applied to *heterogeneous* lists and thus deduce that the application of reverse to a list of type, say, `[ Int Bool* Char+ ]` yields a result of type `[ Char+ Bool* Int ]`). As such it is orthogonal to the kind of polymorphism presented here, and both can and should coexist in a same language.

[7] Alain Frisch's PhD. thesis [9] describes two algorithms that improve over the simple saturation-based strategy described in Section 3.4. They are used both in ℂDuce compiler and in the prototype we implemented to check the subtyping relation presented in this work.

3. Formal development

In this section we describe the technical development that supports the results we exposed in the previous section. We (strongly) suggest readers to skip this section on first reading and directly jump to the conclusions in Section 4. The prose is reduced to the strict necessary, and limited to explain points that were not dealt with in the previous section.

3.1 Types

Definition 3.1. (Types) *Consider a countable set of type variables $\mathcal{V}$ ranged over by α and a finite set of basic (or constant) types ranged over by b. A* type *is a regular tree coinductively produced by the following grammar*

$$t ::= \alpha \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

*and in which every infinite branch contains infinitely many occurrences of atoms (*ie*, either a type variable or the immediate application of a type constructor: basic, product, arrow)*

We write $t_1 \backslash t_2$, $t_1 {\wedge} t_2$, and $\mathbb{1}$ respectively as abbreviation for $t_1 {\wedge} \neg t_2$, $\neg(\neg t_1 {\vee} \neg t_2)$, and $\neg\mathbb{0}$.

The condition on infinite branches bars out ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). It also ensures that the binary relation $\triangleright \subseteq \mathcal{T}^2$ defined by $t_1 \vee t_2 \triangleright t_i$, $\neg t \triangleright t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}$ that we will use without any further explicit reference to the relation.

Notation. *Let t be a type. We use* $\mathsf{var}(t)$ *to denote the set of type variables occurring in t and by* $\mathsf{tlv}(t)$ *(toplevel variables) all the variables of t that have at least one occurrence not under a constructor, that is:* $\mathsf{tlv}(\alpha) = \{\alpha\}$, $\mathsf{tlv}(\neg t) = \mathsf{tlv}(t)$, $\mathsf{tlv}(t \vee s) = \mathsf{tlv}(t) \cup \mathsf{tlv}(s)$, *and* $\mathsf{tlv}(t) = \varnothing$ *otherwise. We say that t is* ground *or* closed *if and only if* $\mathsf{var}(t)$ *is empty.*

3.2 Subtyping

Definition 3.2 (Set-Theoretic Interpretation). *A* set-theoretic interpretation *of $\mathcal{T}$ is given by a set $\mathcal{D}$ and a function $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ such that, for all $t_1, t_2, t \in \mathcal{T}$, $\alpha \in \mathcal{V}$ and $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$: $[\![t_1 \vee t_2]\!]\eta = [\![t_1]\!]\eta \cup [\![t_2]\!]\eta$, $[\![\neg t]\!]\eta = \mathcal{D} \setminus [\![t]\!]\eta$, $[\![\alpha]\!]\eta = \eta(\alpha)$, and $[\![\mathbb{0}]\!]\eta = \varnothing$.*

Definition 3.3. (Subtyping Relation) *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. We define the* subtyping *relation $\leq_{[\![\,]\!]} \subseteq \mathcal{T}^2$ as follows:*

$$t \leq_{[\![\,]\!]} s \Leftrightarrow \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t]\!]\eta \subseteq [\![s]\!]\eta$$

We write $t \leq s$ when the interpretation $[\![\_]\!]$ is clear from the context.

Lemma 3.4. *Let t, s be two types, then: $t \leq_{[\![\,]\!]} s \Leftrightarrow \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t \wedge \neg s]\!]\eta = \varnothing$.*

Proof.

$$t \leq_{[\![\,]\!]} s \Leftrightarrow \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t]\!]\eta \subseteq [\![s]\!]\eta$$
$$\Leftrightarrow \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t]\!]\eta \cap (\mathcal{D} \setminus [\![s]\!]\eta) = \varnothing$$
$$\Leftrightarrow \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t \wedge \neg s]\!]\eta = \varnothing.$$

□

For each basic type b, we assume there is a fixed set of constants $\mathbb{B}(b) \subseteq \mathscr{C}$ whose elements are called constants of type b. For two basic types b_1, b_2, the sets $\mathbb{B}(b_i)$ can have a non-empty intersection.

If, as suggested in Section 2, we interpret extensionally an arrow $t_1 \to t_2$ as $\mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$ (precisely as $\mathscr{P}(\mathcal{D}^2 \setminus ([\![t_1]\!] \times (\mathcal{D} \setminus [\![t_2]\!]))))$, then every function type is a subtype of $\mathbb{1} \to \mathbb{1}$. We do not want such a property to hold because, otherwise, we could subsume every function to a function that accepts every value and, therefore, every application of a well-typed function to a well-typed argument would be well-typed, independently from the types of the function and of the argument. For example, if, say, succ: $\mathtt{Int} \to \mathtt{Int}$, then we could deduce succ: $\mathbb{1} \to \mathbb{1}$ and then succ(true) would have type $\mathbb{1}$. To avoid this problem we introduce an explicit type error Ω and use it to define function spaces:

Definition 3.5. *If D is a set and X, Y are subsets of D, we write D_Ω for $D + \{\Omega\}$ and define $X \to Y$ as:*

$$X \to Y \stackrel{\text{def}}{=} \{f \subseteq D \times D_\Omega \mid \forall (d, d') \in f.\ d \in X \Rightarrow d' \in Y\}$$

This is used in the definition of the extensional interpretation:

Definition 3.6 (Extensional Interpretation). *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. Its associated* extensional interpretation *is the unique function $\mathbb{E}(\_) : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathbb{E}\mathcal{D})$ where $\mathbb{E}\mathcal{D} = \mathcal{D} + \mathcal{D}^2 + \mathscr{P}(\mathcal{D} \times \mathcal{D}_\Omega)$, defined as follows:*

$$\begin{aligned}
\mathbb{E}(\alpha)\eta &= \eta(\alpha) \subseteq \mathcal{D} \\
\mathbb{E}(b)\eta &= \mathbb{B}(b) \subseteq \mathcal{D} \\
\mathbb{E}(t_1 \times t_2)\eta &= [\![t_1]\!]\eta \times [\![t_2]\!]\eta \subseteq \mathcal{D}^2 \\
\mathbb{E}(t_1 \to t_2)\eta &= [\![t_1]\!]\eta \to [\![t_2]\!]\eta \subseteq \mathscr{P}(\mathcal{D} \times \mathcal{D}_\Omega) \\
\mathbb{E}(\mathbb{0})\eta &= \varnothing \\
\mathbb{E}(t_1 \vee t_2)\eta &= \mathbb{E}(t_1)\eta \cup \mathbb{E}(t_2)\eta \\
\mathbb{E}(\neg t)\eta &= \mathbb{E}\mathcal{D} \setminus \mathbb{E}(t)\eta
\end{aligned}$$

The definition of set-theoretic model is then as expected:

Definition 3.7 (Foundation, Convexity, and Models). *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. It is*

1. convex *if it satisfies equation (8) for all finite choices of types $t_1, ..., t_n$;*
2. structural *if $\mathcal{D}^2 \subseteq \mathcal{D}$, $[\![t_1 \times t_2]\!]\eta = [\![t_1]\!]\eta \times [\![t_2]\!]\eta$ and the relation on $\mathcal{D}$ induced by $(d_1, d_2) \blacktriangleright d_i$ is Noetherian;*
3. *a* model *if it induces the same subtyping relation as its associated extensional interpretation, that is: $\forall t \in \mathcal{T} . \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t]\!]\eta = \varnothing \iff \mathbb{E}(t)\eta = \varnothing$. A model is* convex *if its set-theoretic interpretation is convex; it is* well-founded *if it induces the same subtyping relation as a structural set-theoretic interpretation.*

From now on we consider only *well-founded convex* models. We already explained the necessity of the notion of convexity we introduced in §2.5. The notion of well-founded model was introduced in §4.3 of [11]. Intuitively, well-founded models are models that contain only values that are finite (*eg*, in a well-founded model the type $\mu x.(x \times x)$—*ie*, the type that "should" contain all and only infinite binary trees—is empty). This fits the practical motivations of this paper, since XML documents—*ie*, values—are *finite* trees.

3.3 Properties of the subtyping relation

We write $\mathscr{A}_{\mathsf{fun}}$ for atoms of the form $t_1 \to t_2$, $\mathscr{A}_{\mathsf{prod}}$ for atoms of the form $t_1 \times t_2$, $\mathscr{A}_{\mathsf{basic}}$ for basic types, and $\mathscr{A}$ for $\mathscr{A}_{\mathsf{fun}} \cup \mathscr{A}_{\mathsf{prod}} \cup \mathscr{A}_{\mathsf{basic}}$. Therefore $\mathcal{V} \cup \mathscr{A} \cup \{\mathbb{0}, \mathbb{1}\}$ denotes the set of all atoms. Henceforth, we will disregard the atoms $\mathbb{0}$ and $\mathbb{1}$ since they can be straightforwardly eliminated during the algorithmic treatment of subtyping.

Definition 3.8. (Normal Form) *A (disjunctive)* normal form *τ is a finite set of pairs of finite sets of atoms, that is, an element of $\mathscr{P}_f(\mathscr{P}_f(\mathscr{A} \cup \mathcal{V}) \times \mathscr{P}_f(\mathscr{A} \cup \mathcal{V}))$ (where $\mathscr{P}_f(.)$ denotes the finite powerset). Moreover, we call an element of $\mathscr{P}_f(\mathscr{A} \cup \mathcal{V}) \times \mathscr{P}_f(\mathscr{A} \cup \mathcal{V})$ a* single normal form*. If $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ is an arbitrary set-theoretic interpretation, τ a normal form and η an assignment, we define $[\![\tau]\!]\eta$ as:*

$$[\![\tau]\!]\eta = \bigcup_{(P,N) \in \tau} \bigcap_{a \in P} [\![a]\!]\eta \cap \bigcap_{a \in N} (\mathcal{D} \setminus [\![a]\!]\eta)$$

(with the convention that an intersection over an empty set is taken to be $\mathcal{D}$).

Lemma 3.9. *For every type $t \in \mathcal{T}$, it is possible to compute a normal form $\mathcal{N}(t)$ such that for every set-theoretic interpretation $[\![\_]\!]$ and assignment η, $[\![t]\!]\eta = [\![\mathcal{N}(t)]\!]\eta$.*

Proof. We can define two functions $\mathcal{N}$ and $\mathcal{N}'$, both from types to $\mathscr{P}_f(\mathscr{P}_f(\mathscr{A} \cup \mathcal{V}) \times \mathscr{P}_f(\mathscr{A} \cup \mathcal{V}))$, by mutual induction over types:

$$\begin{aligned}
\mathcal{N}(\mathbb{0}) &= \varnothing \\
\mathcal{N}(a) &= \{(\{a\}, \varnothing)\} && \text{for } a \in \mathscr{A} \cup \mathcal{V} \\
\mathcal{N}(t_1 \vee t_2) &= \mathcal{N}(t_1) \cup \mathcal{N}(t_2) \\
\mathcal{N}(\neg t) &= \mathcal{N}'(t)
\end{aligned}$$

$$\begin{aligned}
\mathcal{N}'(\mathbb{0}) &= \{(\varnothing, \varnothing)\} \\
\mathcal{N}'(a) &= \{(\varnothing, \{a\})\} && \text{for } a \in \mathscr{A} \cup \mathcal{V} \\
\mathcal{N}'(t_1 \vee t_2) &= \{(P_1 \cup P_2, N_1 \cup N_2) \mid (P_i, N_i) \in \mathcal{N}'(t_i), i{=}1,2\} \\
\mathcal{N}'(\neg t) &= \mathcal{N}(t)
\end{aligned}$$

Then we check the following property by induction over the type t:

$$[\![t]\!]\eta = [\![\mathcal{N}(t)]\!]\eta = \mathcal{D} \setminus [\![\mathcal{N}'(t)]\!]\eta$$

□

For instance, consider the type $t = a_1 \wedge (a_3 \vee \neg a_2)$ where a_1, a_2, and a_3 are any atoms. Then $\mathcal{N}(t) = \{(\{a_1, a_3\}, \varnothing), (\{a_1\}, \{a_2\})\}$. This corresponds to the fact that for every set-theoretic interpretation and semantic assignment $\mathcal{N}(t)$, t, and $(a_1 \wedge a_3) \vee (a_1 \wedge \neg a_2)$ have the same denotation.

Note that the converse result is true as well: for any normal form τ, we can find a type t such that for every set-theoretic interpretation η and semantic assignment $[\![\_]\!]$, $[\![t]\!]\eta = [\![\tau]\!]\eta$. Normal forms are thus simply a different, but handy, syntax for types. In particular, we can rephrase in Definition 3.7 the condition for a set-theoretic interpretation to be a model as: for any normal form τ, $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![\tau]\!]\eta = \varnothing \Leftrightarrow \mathbb{E}(\tau)\eta = \varnothing$. For these reasons henceforth we will often confound the notions of types and normal forms, and often speak of the "type" τ, taking the latter as a canonical representation of all the types in $\mathcal{N}^{-1}(\tau)$.

Let $[\![\_]\!]$ be a set-theoretic interpretation. Given a normal form τ, we are interested in comparing the assertions $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \mathbb{E}(\tau)\eta = \varnothing$ and $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![\tau]\!]\eta = \varnothing$. Clearly, the equation $\forall \eta \in$

$\mathscr{P}(\mathcal{D})^{\mathcal{V}}.\, \mathbb{E}(\tau)\eta = \varnothing$ is equivalent to:

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \forall (P,N) \in \tau . \bigcap_{a\in P} \mathbb{E}(a)\eta \subseteq \bigcup_{a\in N} \mathbb{E}(a)\eta \quad (14)$$

Let us write $\mathbb{E}^{\mathsf{basic}}\mathcal{D} = \mathcal{D}$, $\mathbb{E}^{\mathsf{prod}}\mathcal{D} = \mathcal{D}^2$ and $\mathbb{E}^{\mathsf{fun}}\mathcal{D} = \mathscr{P}(\mathcal{D} \times \mathcal{D}_\Omega)$. Then we have $\mathbb{E}\mathcal{D} = \bigcup_{u\in U} \mathbb{E}^u\mathcal{D}$ where $U = \{\mathsf{basic}, \mathsf{prod}, \mathsf{fun}\}$. Thus we can rewrite Inequality (14) as:

$$\begin{array}{c} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \forall u \in U . \forall (P,N) \in \tau . \\ \bigcap_{a\in P} (\mathbb{E}(a)\eta \cap \mathbb{E}^u\mathcal{D}) \subseteq \bigcup_{a\in N} (\mathbb{E}(a)\eta \cap \mathbb{E}^u\mathcal{D}) \end{array} \quad (15)$$

For an atom $a \in \mathscr{A}$, we have $\mathbb{E}(a)\eta \cap \mathbb{E}^u\mathcal{D} = \varnothing$ if $a \notin \mathscr{A}_u$ and $\mathbb{E}(a)\eta \cap \mathbb{E}^u\mathcal{D} = \mathbb{E}(a)\eta$ if $a \in \mathscr{A}_u$. Then we can rewrite Inequality (15) as:

$$\begin{array}{l} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \forall u \in U. \forall (P,N) \in \tau . (P \subseteq \mathscr{A}_u \cup \mathcal{V}) \Rightarrow \\ \quad \bigcap_{a\in P\cap\mathscr{A}_u} \mathbb{E}(a)\eta \cap \bigcap_{\alpha\in P\cap\mathcal{V}} (\eta(\alpha) \cap \mathbb{E}^u\mathcal{D}) \subseteq \\ \qquad \bigcup_{a\in N\cap\mathscr{A}_u} \mathbb{E}(a)\eta \cup \bigcup_{\alpha\in N\cap\mathcal{V}} (\eta(\alpha) \cap \mathbb{E}^u\mathcal{D}) \end{array} \quad (16)$$

(where the intersection is taken to be $\mathbb{E}\mathcal{D}$ when $P = \varnothing$). Furthermore, if the same variable occurs both in P and in N, then (16) is trivially satisfied. So we can suppose that $P \cap N \cap \mathcal{V} = \varnothing$. This justifies the simplifications made in *Step 3* of the subtyping algorithm, that is the simplification of mixed single normal forms.

Step 4, the elimination of negated toplevel variables, is justified by the following lemma:

Lemma 3.10. *Let P, N be two finite subsets of atoms and α an arbitrary type variable, then we have*

$$\begin{array}{l} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \bigcap_{a\in P} \mathbb{E}(a)\eta \subseteq \bigcup_{a\in N} \mathbb{E}(a)\eta \cup \eta(\alpha) \Leftrightarrow \\ \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \bigcap_{a\in P} \mathbb{E}(\theta_\alpha^{\neg\beta}(a))\eta \cap \eta(\beta) \subseteq \bigcup_{a\in N} \mathbb{E}(\theta_\alpha^{\neg\beta}(a))\eta \end{array}$$

where β is a fresh variable and $\theta_\alpha^{\neg\beta}(a) = a\{^{\neg\beta}/_{\alpha}\}$

Proof. Straightforward application of set theory. □

Note that Lemma 3.10 only deals with one type variable, but it is trivial to generalize this lemma to multiple type variables (the same holds for Lemmas 3.13 and 3.14).

Using Lemma 3.10, we can rewrite Inequality (16) as:

$$\begin{array}{l} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \forall u \in U . \forall (P,N) \in \tau . (P \subseteq \mathscr{A}_u \cup \mathcal{V}) \Rightarrow \\ \bigcap_{a\in P\cap\mathscr{A}_u} \mathbb{E}(a)\eta \cap \bigcap_{\alpha\in P\cap\mathcal{V}} (\eta(\alpha) \cap \mathbb{E}^u\mathcal{D}) \subseteq \bigcup_{a\in N\cap\mathscr{A}_u} \mathbb{E}(a)\eta \end{array} \quad (17)$$

since we can assume $N \cap \mathcal{V} = \varnothing$.

Next, we justify *Step 5* of the algorithm, that is the elimination of the toplevel variables. In (17) this corresponds to eliminating the variables in $P \cap \mathcal{V}$. When $u = \mathsf{basic}$ this can be easily done since all variables (which can appear only at top-level) can be straightforwardly removed. Indeed, notice that if s and t are closed types then $s \wedge \alpha \leq t$ if and only if $s \leq t$. Since unions and intersections of basic types are closed, then we have the following lemma

Lemma 3.11. *Let P, N be two finite subsets of $\mathscr{A}_{\mathsf{basic}}$, X a finite set of variables. Then*

$$\begin{array}{l} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \bigcap_{b\in P} \mathbb{E}(b)\eta \cap \bigcap_{\alpha\in X} (\eta(\alpha) \cap \mathscr{C}) \subseteq \bigcup_{b\in N} \mathbb{E}(b)\eta \\ \Leftrightarrow \bigcap_{b\in P} \mathbb{B}(b) \subseteq \bigcup_{b\in N} \mathbb{B}(b) \end{array}$$

(with the convention $\bigcap_{a\in\varnothing} \mathbb{E}(a)\eta = \mathscr{C}$)

The justification of *Step 5* for $u = \mathsf{prod}$ is given by Lemma 3.13. In order to prove it we need to prove a substitution lemma for the extensional interpretation.

Lemma 3.12. *Let $[\![\_]\!]$ be a well-founded model and $\mathbb{E}(\_)$ its associated extensional interpretation. For all types t, t', variable α, and assignment η, if $\mathbb{E}(t')\eta = \eta(\alpha)$, then $\mathbb{E}(t\{^{t'}/_{\alpha}\})\eta = \mathbb{E}(t)\eta$.*

Proof. Consider a type t. Then $\mathbb{E}(t\{^{t'}/_{\alpha}\})\eta = \mathbb{E}(t)\eta$ holds, if and only if

$$\forall d \in \mathcal{D} . d \in \mathbb{E}(t\{^{t'}/_{\alpha}\})\eta \Leftrightarrow d \in \mathbb{E}(t)\eta$$

Since the model is well-founded, we can apply the induction on d. Meanwhile, we also use the Noetherian order relation $\triangleright$ on t (See Section 3.1). That is, we prove this statement by induction on (d, t). In the following only the direction "⇐" is proved, the same to the other direction "⇒".

Case β**:** if $\beta \neq \alpha$, the result holds straightforwardly; Otherwise, the result also holds since $\mathbb{E}(t')\eta = \eta(\alpha)$ holds.

Case b **or** $\mathbb{0}$**:** trivially since $\alpha \notin \mathsf{var}(t)$.

Case $t_1 \vee t_2$**:** as $d \in \mathbb{E}(t_1 \vee t_2)\eta$, we have either $d \in \mathbb{E}(t_1)\eta$ or $d \in \mathbb{E}(t_2)\eta$. Assume $d \in \mathbb{E}(t_1)\eta$, then by induction, we have $d \in \mathbb{E}(t_1\{^{t'}/_{\alpha}\})\eta$. Thus the result follows.

Case $t_1 \wedge t_2$**:** similarly to Case $t_1 \vee t_2$.

Case $\neg t_1$**:** assume $d \notin \mathbb{E}(\neg t_1\{^{t'}/_{\alpha}\})\eta$, that is $d \in \mathbb{E}(t_1\{^{t'}/_{\alpha}\})\eta$, then by induction, we have $d \in \mathbb{E}(t_1)\eta$, which contradicts $d \in \mathbb{E}(\neg t_1)\eta$. Therefore, the result follows.

Case $t_1 \times t_2$**:** then $d = (d_1, d_2)$. Since $\mathbb{E}(t_1 \times t_2)\eta = \mathbb{E}(t_1)\eta \times \mathbb{E}(t_2)\eta$, we have $d_i \in \mathbb{E}(t_i)\eta$. By induction, we get $d_i \in \mathbb{E}(t_i\{^{t'}/_{\alpha}\})\eta$. Thus, $(d_1, d_2) \in \mathbb{E}((t_1 \times t_2)\{^{t'}/_{\alpha}\})$.

Case $t_1 \to t_2$**:** then $d = \{(d_1, d'_1), \ldots, (d_n, d'_n)\}$. Considering any (d_i, d'_i), either both $d_i \in \mathbb{E}(t_1)\eta$ and $d'_i \in \mathbb{E}(t_2)\eta$ hold or $d_i \notin \mathbb{E}(t_1)\eta$ holds. By induction, we have either both $d_i \in \mathbb{E}(t_1\{^{t'}/_{\alpha}\})\eta$ and $d'_i \in \mathbb{E}(t_2\{^{t'}/_{\alpha}\})\eta$ or $d_i \notin \mathbb{E}(t_1\{^{t'}/_{\alpha}\})\eta$. Thus, $d \in \mathbb{E}((t_1 \times t_2)\{^{t'}/_{\alpha}\})$.

□

Lemma 3.13. *Let $(\mathcal{D}, [\![.]\!])$ be a well-founded model, P, N two finite subsets of $\mathscr{A}_{\mathsf{prod}}$ and α an arbitrary type variable.*

$$\begin{array}{l} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \bigcap_{a\in P} \mathbb{E}(a)\eta \cap (\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D}) \subseteq \bigcup_{a\in N} \mathbb{E}(a)\eta \Leftrightarrow \\ \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . \bigcap_{a\in P} \mathbb{E}(\theta_\alpha^{\times}(a))\eta \cap \mathbb{E}(\alpha_1 \times \alpha_2)\eta \subseteq \bigcup_{a\in N} \mathbb{E}(\theta_\alpha^{\times}(a))\eta \end{array}$$

where $\theta_\alpha^{\times}(a) = a\{^{(\alpha_1\times\alpha_2)\vee\alpha}/_{\alpha}\}$ and α_1, α_2 are fresh variables.

Proof. "⇐" direction: consider a generic assignment η, and suppose we have $\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D} = \bigcup_{i\in I}(S_1^i \times S_2^i)$ where S_j^i are subsets of $\mathcal{D}$ and I may be infinite (notice that every intersection with $\mathcal{D}^2$ can be expressed as a infinite union of products: at the limit singleton products can be used). If $|I| = 0$, that is, $\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D} = \varnothing$, clearly, we have

$$\bigcap_{a\in P} \mathbb{E}(a)\eta \cap (\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D}) = \varnothing \subseteq \bigcup_{a\in N} \mathbb{E}(a)\eta$$

Assume that $|I| > 0$. Then for each $(S_1^i \times S_2^i)$, we construct another assignment η^i defined as $\eta^i = \eta \oplus \{^{S_1^i}/_{\alpha_1}, ^{S_2^i}/_{\alpha_2}\}$, where $\oplus$ denotes both function extension and redefinition. Then, we have

$$\begin{aligned}&\mathbb{E}((\alpha_1{\times}\alpha_2)\vee\alpha)\eta^i\\&\quad=(\eta^i(\alpha_1)\times\eta^i(\alpha_2))\vee\eta^i(\alpha) && \text{by definition of } \mathbb{E}(\_)\\&\quad=(S_1^i{\times}S_2^i)\vee\eta(\alpha) && \text{by definition of } \eta^i\\&\quad=\eta(\alpha) && \text{since } \textstyle\bigcup_{i\in I}(S_1^i{\times}S_2^i)\subseteq\eta(\alpha)\\&\quad=\eta^i(\alpha) && \text{by definition of } \eta^i\end{aligned}$$

We can thus apply Lemma 3.12 and for any term t deduce that $\mathbb{E}(t\{^{(\alpha_1\times\alpha_2)\vee\alpha}/_{\alpha}\})\eta^i=\mathbb{E}(t)\eta^i$. In particular, for all $a\in P\cup N$ we have $\mathbb{E}(\theta_\alpha^\times(a))\eta^i\stackrel{\text{def}}{=}\mathbb{E}(a\{^{(\alpha_1\times\alpha_2)\vee\alpha}/_{\alpha}\})\eta^i=\mathbb{E}(a)\eta^i$. Now, notice that η and η^i differ only for the interpretation of α_1 and α_2. Since these variables are fresh, then they do not belong to $\mathsf{var}(a)$, and therefore $\mathbb{E}(a)\eta^i=\mathbb{E}(a)\eta$. This allows us to conclude that

$$\forall a\in(P\cup N)\;.\;\mathbb{E}(\theta_\alpha^\times(a))\eta^i=\mathbb{E}(a)\eta$$

Therefore,

$$\begin{aligned}&\forall i{\in}I(\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^\times(a))\eta^i\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\subseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^\times(a))\eta^i)\\&\Rightarrow\forall i{\in}I(\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^\times(a))\eta^i\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\cap\bigcap_{a\in N}\overline{\mathbb{E}(\theta_\alpha^\times(a))\eta^i}\subseteq\varnothing)\\&\Rightarrow\bigcup_{i\in I}\left((\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^\times(a))\eta^i\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\cap\bigcap_{a\in N}\overline{\mathbb{E}(\theta_\alpha^\times(a))\eta^i})\right)\subseteq\varnothing\\&\Rightarrow\bigcup_{i\in I}\left((\bigcap_{a\in P}\mathbb{E}(a)\eta\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\cap\bigcap_{a\in N}\overline{\mathbb{E}(a)\eta})\right)\subseteq\varnothing\\&\Rightarrow\bigcap_{a\in P}\mathbb{E}(a)\eta\cap(\bigcup_{i\in I}\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i)\cap\bigcap_{a\in N}\overline{\mathbb{E}(a)\eta}\subseteq\varnothing\\&\Rightarrow\bigcap_{a\in P}\mathbb{E}(a)\eta\cap(\bigcup_{i\in I}(S_1^i{\times}S_2^i))\cap\bigcap_{a\in N}\overline{\mathbb{E}(a)\eta}\subseteq\varnothing\\&\Rightarrow\bigcap_{a\in P}\mathbb{E}(a)\eta\cap(\eta(\alpha)\cap\mathbb{E}^{\mathsf{prod}}\mathcal{D})\cap\bigcap_{a\in N}\overline{\mathbb{E}(a)\eta}\subseteq\varnothing\\&\Rightarrow\bigcap_{a\in P}\mathbb{E}(a)\eta\cap(\eta(\alpha)\cap\mathbb{E}^{\mathsf{prod}}\mathcal{D})\subseteq\bigcup_{a\in N}\mathbb{E}(a)\eta\end{aligned}$$

This proves the result.

"$\Rightarrow$" direction: this direction is rather obvious since if a type is empty, then so is every instance of it. In particular, suppose there exists an assignment η such that

$$\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^\times(a))\eta\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta\subseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^\times(a))\eta$$

does not hold. Then, for this assignment η we have

$$\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^\times(a))\eta\cap(\mathbb{E}(\alpha_1{\times}\alpha_2)\eta)\nsubseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^\times(a))\eta$$

and a fortiori

$$\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^\times(a))\eta\cap(\mathbb{E}(\alpha_1{\times}\alpha_2)\eta\cup\eta(\alpha))\nsubseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^\times(a))\eta$$

That is,

$$\mathbb{E}(\theta_\alpha^\times(\bigwedge_{a\in P}a\wedge\alpha\wedge\bigwedge_{a\in N}\neg a))\eta\neq\varnothing$$

therefore

$$\bigcap_{a\in P}\mathbb{E}(a)\eta'\cap(\eta'(\alpha)\cap\mathbb{E}^{\mathsf{prod}}\mathcal{D})\subseteq\bigcup_{a\in N}\mathbb{E}(a)\eta'$$

doesn't hold for $\eta'=\eta\oplus\{^{\mathbb{E}((\alpha_1\times\alpha_2)\vee\alpha)\eta}/_{\alpha}\}$ which contradicts the hypothesis. □

The case for $u=\mathsf{fun}$ is trickier because $\mathbb{1}\to\mathbb{0}$ is contained in every arrow type, and therefore sets of arrows that do not contain it must be explicitly checked. If, analogously to $u=\mathsf{prod}$, we used just $\{^{(\alpha_1\to\alpha_2)\vee\alpha}/_{\alpha}\}$, then the subtypes of $\mathbb{0}\to\mathbb{1}$ that do not contain $\mathbb{1}\to\mathbb{0}$ would never be assigned to α by the algorithm and, thus, never checked. Therefore, $\{^{((\alpha_1\to\alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\vee\alpha}/_{\alpha}\}$ must be checked, as well.

Lemma 3.14. *Let $(\mathcal{D},[\![.]\!])$ be a well-founded model, P,N two finite subsets of $\mathscr{A}_{\mathsf{fun}}$ and α an arbitrary type variable. Then*

$$\forall\eta{\in}\mathscr{P}(\mathcal{D})^{\mathcal{V}}.\bigcap_{a\in P}\mathbb{E}(a)\eta\cap(\eta(\alpha)\cap\mathbb{E}^{\mathsf{fun}}\mathcal{D})\subseteq\bigcup_{a\in N}\mathbb{E}(a)\eta\Leftrightarrow$$

$$\forall\eta{\in}\mathscr{P}(\mathcal{D})^{\mathcal{V}}.\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^{\to}(a))\eta\cap\mathbb{E}(\alpha_1{\to}\alpha_2)\eta\subseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^{\to}(a))\eta$$

and $$\forall\eta{\in}\mathscr{P}(\mathcal{D})^{\mathcal{V}}.\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta\cap\mathbb{E}((\alpha_1{\to}\alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\eta\subseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta$$

where α_1, α_2 are fresh variables, $\theta_\alpha^{\to}(a)=a\{^{(\alpha_1\to\alpha_2)\vee\alpha}/_{\alpha}\}$, and $\theta_\alpha^{\leadsto}(a)=a\{^{((\alpha_1\to\alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\vee\alpha}/_{\alpha}\}$.

Proof. "$\Leftarrow$" direction: suppose that there exists one assignment η such that

$$\bigcap_{a\in P}\mathbb{E}(a)\eta\cap(\mathbb{E}(\alpha)\eta\cap\mathbb{E}^{\mathsf{fun}}\mathcal{D})\subseteq\bigcup_{a\in N}\mathbb{E}(a)\eta$$

does not hold. Then for this assignment, there exists at least an element d such that

$$d\in(\bigcap_{a\in P}\mathbb{E}(a)\eta\setminus\bigcup_{a\in N}\mathbb{E}(a)\eta)\cap(\mathbb{E}(\alpha)\eta\cap\mathbb{E}^{\mathsf{fun}}\mathcal{D})$$

If one of these elements d is such that $d\in\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta$, then $|N|=0$: indeed since $\mathbb{1}{\to}\mathbb{0}$ is contained in every arrow type, then subtracting any arrow type (*ie*, $|N|>0$) would remove all the elements of $\mathbb{1}{\to}\mathbb{0}$. Clearly, we have

$$d\in(\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^{\to}(a))\eta\setminus\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^{\to}(a))\eta)\cap\mathbb{E}(\alpha_1\to\alpha_2)\eta$$

Indeed since $|N|=0$ the set above is an intersection of arrow types, and since they all contain $\mathbb{1}{\to}\mathbb{0}$, they all contain d as well. This contradicts the premise, therefore, the result follows. (Note that when doing the substitution for α we don't need to consider the case that $\mathbb{E}(\alpha)\eta$ contains a part of $\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta$ since there always exists an element d belonging to such a part.)
Otherwise, assume that $|N|>0$ and therefore $d\notin\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta$. Since $d\in(\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^{\to}(a))\eta\setminus\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^{\to}(a))\eta)$ then we have

$$\bigcap_{a\in P}\mathbb{E}(a)\eta\nsubseteq\bigcup_{a\in N}\mathbb{E}(a)\eta.$$

Let η_1 be an assignment defined as $\eta_1=\eta\oplus\{^{\mathcal{D}}/_{\alpha_1},{}^{\varnothing}/_{\alpha_2}\}$. Then, we have $\mathbb{E}(\alpha_1\to\alpha_2)\eta_1=\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta_1$. Therefore:

$$\begin{aligned}&\mathbb{E}(\theta_\alpha^{\leadsto}(\alpha))\eta_1\\&=(\mathbb{E}(\alpha_1\to\alpha_2)\eta_1\setminus\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta_1)\cup\eta_1(\alpha)\\&\quad\text{(by definition of }\mathbb{E}(\_))\\&=\eta_1(\alpha)\quad\text{(by definition of }\eta_1)\end{aligned}$$

We thus apply Lemma 3.12 and for any $a\in P\cup N$, we deduce that $\mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta_1=\mathbb{E}(a)\eta_1=\mathbb{E}(a)\eta$. From this, we infer that

$$\bigcap_{a\in P}\mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta_1\nsubseteq\bigcup_{a\in N}\mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta_1$$

By an application of Lemma 6.7 in [11], we have

$$\forall t_1^0 \to t_2^0 \in N.\exists P' \subseteq P.$$
$$\begin{cases} [\![\theta_\alpha^{\leadsto}(t_1^0 \setminus \bigvee_{t_1\to t_2\in P'} t_1)]\!]\eta_1 \neq \varnothing \\ \text{and} \\ P' = P \text{ or } [\![\theta_\alpha^{\leadsto}(\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \setminus t_2^0)]\!]\eta_1 \neq \varnothing \end{cases}$$

Thus there exist at least an element in $[\![\theta_\alpha^{\leadsto}(t_1^0 \setminus \bigvee_{t_1\to t_2\in P'} t_1)]\!]\eta_1$ and, if $P \neq P'$, an element in $[\![\theta_\alpha^{\leadsto}(\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \setminus t_2^0)]\!]\eta_1$. The next step is to build a new assignment η' such that $[\![\theta_\alpha^{\leadsto}(t_1^0 \setminus \bigvee_{t_1\to t_2\in P'} t_1 \vee \alpha_1)]\!]\eta'$ contains an element and, if $P \neq P'$, $[\![\theta_\alpha^{\leadsto}((\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \wedge \alpha_2) \setminus t_2^0)]\!]\eta'$ contains an element.
To do so we invoke the procedure explore_pos defined in the proof of Lemma 3.29 (this procedure was defined for the proof of that lemma, and it returns also the elements that inhabit the types, which are here useless. We do not repeat its definition here). Let $VS = \bigcup_{a\in P\cup N} \mathsf{var}(a) \cup \{\alpha_1, \alpha_2\}$. For each $\beta \in VS$, we construct a finite set s_β which is initialized as an empty set and appended some elements during the processing of explore_pos. Thanks to the infinite support, to build an element in $\theta_\alpha^{\leadsto}(t_1^0 \setminus \bigvee_{t_1\to t_2\in P'} t_1 \vee \alpha_1)$ is similar to build an element in $\theta_\alpha^{\leadsto}(t_1^0 \setminus \bigvee_{t_1\to t_2\in P'} t_1)$ (it has proved that such an element exists). And to build an element in $\theta_\alpha^{\leadsto}((\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \wedge \alpha_2) \setminus t_2^0)$, is to build an element in $\theta_\alpha^{\leadsto}(\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \setminus t_2^0)$ first and then append this element to s_{α_2}. In the end, we define a new (finite) assignment as $\eta' = \{{}^{s_\beta}/_{\beta}, \ldots\}$ for $\beta \in VS$. (If any type contains infinite product types it is also possible to construct such an assignment by Lemma 3.30). Therefore, under the assignment η', we get

$$\forall t_1^0 \to t_2^0 \in N.\exists P' \subseteq P.$$
$$\begin{cases} [\![\theta_\alpha^{\leadsto}(t_1^0 \setminus \bigvee_{t_1\to t_2\in P'} t_1 \vee \alpha_1)]\!]\eta' \neq \varnothing \\ \text{and} \\ [\![\theta_\alpha^{\leadsto}((\bigwedge_{t_1\to t_2\in P\setminus P'} t_2 \wedge \alpha_2) \setminus t_2^0)]\!]\eta' \neq \varnothing \end{cases}$$
$$\text{or}$$
$$\begin{cases} [\![\theta_\alpha^{\leadsto}(t_1^0 \setminus (\bigvee_{t_1\to t_2\in P'} t_1 \vee \alpha_1))]\!]\eta' \neq \varnothing \\ \text{and} \\ P' = P \end{cases}$$

By an application of Lemma 6.7 in [11] again, we conclude that

$$\bigcap_{a\in P} \mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta' \cap \mathbb{E}(\alpha_1 \to \alpha_2)\eta' \not\subseteq \bigcup_{a\in N} \mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta'$$

Since $|N| > 0$, then $\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta'$ is contained in $\bigcup_{a\in N} \mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta'$. Thus removing it from the the left hand side does not change the result, which allows us to conclude that:

$$\bigcap_{a\in P} \mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta' \cap \mathbb{E}((\alpha_1 \to \alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\eta' \not\subseteq \bigcup_{a\in N} \mathbb{E}(\theta_\alpha^{\leadsto}(a))\eta'$$

which again contradicts the hypothesis. Therefore the result follows as well.

"⇒" direction: similarly to the "⇒" direction in the proof of Lemma 3.13. □

The check of the substitution $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee {}^{\alpha}/_{\alpha}\}$ is necessary to the soundness of the algorithm. To see it consider the following relation (found by Nils Gesbert): $((\alpha \to \beta) \wedge \alpha) \leq ((\mathbb{1} \to \mathbb{0}) \to \beta)$. If the check above were not performed, then the algorithm would return that the relation holds, while it does not. In order to see that it does not hold, consider a type $t_1 = (\mathbb{1} \to \mathbb{0}) \wedge \gamma$, where γ is some variable. Clearly, there exists an assignment η_0 such that $[\![t_1]\!]\eta_0$ is nonempty. Let η be another assignment defined as $\eta = \eta_0 \oplus \{{}^{[\![\neg t_1]\!]\eta_0}/_{\alpha}, {}^{\varnothing}/_{\beta}\}$. Next, consider a function $f \in [\![(\alpha \to \beta) \wedge (\neg\alpha \to \mathbb{1})]\!]\eta$, that is, a function that diverges (since $[\![\beta]\!]\eta = \varnothing$) on values in $[\![\alpha]\!]\eta$ (*ie*, $[\![\neg t_1]\!]\eta$). Take f so that it converges on the values in $[\![\neg\alpha]\!]\eta$ (*ie*, $[\![t_1]\!]\eta$). This implies that that $f \notin [\![\neg\alpha]\!]\eta$: indeed, assume $f \in [\![\neg\alpha]\!]\eta$, that is $f \in [\![t_1]\!]\eta_0$, then $f \in [\![\mathbb{1} \to \mathbb{0}]\!]\eta$ and, thus, f would diverge on all values: contradiction (note that $[\![t_1]\!]\eta_0 \neq \varnothing$). Therefore $f \in [\![\alpha]\!]\eta$, and then $f \in [\![(\alpha \to \beta) \wedge \alpha]\!]\eta$. Instead, by construction f does not diverge on $[\![t_1]\!]\eta_0$, which is a nonempty subset of $[\![\mathbb{1} \to \mathbb{0}]\!]\eta_0$. Therefore $f \notin [\![(\mathbb{1} \to \mathbb{0}) \to \beta]\!]\eta$.

If we remove the check of $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee {}^{\alpha}/_{\alpha}\}$ from the algorithm, then the answer of the algorithm would be positive for the relation above since it would just check that all the following four relations hold:

$$\begin{cases} \mathbb{1} \to \mathbb{0} \leq \mathbb{0} \text{ or } \beta \wedge \alpha_2 \leq \beta & (1) \\ \mathbb{1} \to \mathbb{0} \leq (\alpha\{(\alpha_1 \to \alpha_2) \vee {}^{\alpha}/_{\alpha}\}) \text{ or } \alpha_2 \leq \beta & (2) \\ \mathbb{1} \to \mathbb{0} \leq \alpha_1 \text{ or } \beta \leq \beta & (3) \\ \mathbb{1} \to \mathbb{0} \leq \alpha_1 \vee (\alpha\{(\alpha_1 \to \alpha_2) \vee {}^{\alpha}/_{\alpha}\}) & (4) \end{cases}$$

where α_1, α_2 are two fresh variables. Clearly, these four relations hold if (a superset of) $[\![\mathbb{1} \to \mathbb{0}]\!]$ is assigned to α (see the substitution $\{(\alpha_1 \to \alpha_2) \vee {}^{\alpha}/_{\alpha}\}$). However, if there exists a nonempty set $s \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]\eta$ which is not assigned to α (*eg*, the non-empty set $[\![t_1]\!]\eta_0$ defined before), then we should substitute for the occurrences of α that are not at top-level by $((\alpha_1 \to \alpha_2) \setminus ((\mathbb{1} \to \mathbb{0}) \wedge \gamma)) \vee \alpha$ rather than by $(\alpha_1 \to \alpha_2) \vee \alpha$. Thus Case (2) and Case (4) would not hold, and so does not the whole example. Therefore, we need to consider the case that a nonempty subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ is not assigned to α, namely a strict subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ is assigned to α.

Since the interpretation $[\![\mathbb{1} \to \mathbb{0}]\!]$ is infinite, there are infinitely many strict subsets to be considered. Assume there exists a strict subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ that is assigned to α, and there exists an occurrence of α not at top level such that once it moves up to the top level it occurs in the subtyping relation of the form in Lemma 3.14 (otherwise $\eta(\alpha) \cap \mathbb{E}^{\mathsf{fun}}\mathcal{D}$ would be straightforwardly ignored). Since $\mathbb{1} \to \mathbb{0}$ is contained in every arrow type, then either the strict subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ can be ignored if the occurrence is positive, or the subtyping relation does not hold if the occurrence is negative (see Case (2) above). Note that what the strict subset actually is does not matter. Therefore, we just take the empty set into consideration, that is the case of $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee {}^{\alpha}/_{\alpha}\}$. Thus, only two cases — whether $[\![\mathbb{1} \to \mathbb{0}]\!]$ is assigned to α or not — are enough for Lemma 3.14.

As an aside notice that both lemmas above would not hold if the variable α in their statements occurred negated at toplevel, whence the necessity of *Step 4*.

Finally, *Step 6* is justified by the two following lemmas in whose proofs the hypothesis of convexity plays a crucial role:

Lemma 3.15. *Let $(\mathcal{D}, [\![.]\!])$ be a convex set-theoretic interpretation and P, N two finite subsets of $\mathscr{A}_{\mathsf{prod}}$. Then:*

$$\forall\eta. \bigcap_{a\in P} \mathbb{E}(a)\eta \subseteq \bigcup_{a\in N} \mathbb{E}(a)\eta \Leftrightarrow$$

$$\forall N' \subseteq N\ .\ \begin{cases} \forall\eta.[\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1]\!]\eta = \varnothing \\ \vee \\ \forall\eta.[\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2]\!]\eta = \varnothing \end{cases}$$

(with the convention $\bigcap_{a\in\varnothing} \mathbb{E}(a)\eta = \mathbb{E}^{\mathsf{prod}}\mathcal{D}$).

Proof. The Lemma above is a straightforward application of the convexity property and of Lemma 6.4 in [11]. In particular thanks to the latter it is possible to deduce the following equivalence

$$\forall\eta. \bigcap_{a\in P} \mathbb{E}(a)\eta \subseteq \bigcup_{a\in N} \mathbb{E}(a)\eta \Leftrightarrow$$

$$\forall\eta.\forall N' \subseteq N \;.\; \begin{cases} [\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1]\!]\eta = \varnothing \\ \vee \\ [\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2]\!]\eta = \varnothing \end{cases}$$

The result is a straightforward application of the convexity property. □

Lemma 3.16. *Let $(\mathcal{D}, [\![.]\!])$ be a convex set-theoretic interpretation and P, N be two finite subsets of $\mathscr{A}_{\mathsf{fun}}$. Then:*

$$\forall\eta.\bigcap_{a\in P}\mathbb{E}(a)\eta \subseteq \bigcup_{a\in N}\mathbb{E}(a)\eta \Leftrightarrow$$

$$\exists(t_0{\to}s_0){\in}N\,.\,\forall P'{\subseteq}P\,.\, \begin{cases} \forall\eta.[\![t_0 \setminus (\bigvee_{t\to s\in P'} t)]\!]\eta = \varnothing \\ \vee \\ \begin{cases} P \neq P' \\ \wedge \\ \forall\eta.[\![(\bigwedge_{t\to s\in P\setminus P'} s) \setminus s_0]\!]\eta = \varnothing \end{cases} \end{cases}$$

(with the convention $\bigcap_{a\in\varnothing}\mathbb{E}(a)\eta = \mathbb{E}^{\mathsf{fun}}\mathcal{D}$)

Proof. Similarly to previous lemma, the proof can be obtained by a straightforward application of Lemma 6.7 in [11] and the convexity property. □

3.4 Algorithm

The formalization of the subtyping algorithm is done via notion of *simulation* that we borrow from [11] and extend to account for type variables and type instantiation. We use θ to range over (syntactic) substitutions, that is, partial functions from $\mathcal{V}$ to $\mathcal{T}$ (we reserve the meta-variable σ we introduced in Section 2 for ground substitutions).

Definition 3.17. *Given a type $t \in \mathcal{T}$, the application of a substitution θ to t is defined as follows:*

$$\begin{aligned} b\theta &= b \\ (t_1 \times t_2)\theta &= (t_1\theta) \times (t_2\theta) \\ (t_1 \to t_2)\theta &= (t_1\theta) \to (t_2\theta) \\ (t_1 \vee t_2)\theta &= (t_1\theta) \vee (t_2\theta) \\ (\neg t)\theta &= \neg(t\theta) \\ \mathbb{0}\theta &= \mathbb{0} \\ \alpha\theta &= \theta(\alpha) && \text{if } \alpha \in dom(\theta) \\ \alpha\theta &= \alpha && \text{if } \alpha \notin dom(\theta) \end{aligned}$$

This definition is naturally extended to normal forms by applying the substitution to each type in the sets that form the normal form. It is then used to define the set of instances of a type.

Definition 3.18 (Instances). *Given a type $t \in \mathcal{T}$, we define $[t]_\approx$, the* set of instances *of t as:*

$$[t]_\approx \stackrel{\text{def}}{=} \{s \mid \exists\theta : \mathcal{V} \to \mathcal{T}\,.\, t\theta = s\}$$

Definition 3.19 (Simulation). *Let $\mathscr{S}$ be an arbitrary set of normal forms. We define another set of normal forms $\mathbb{E}\mathscr{S}$ as*

$$\{\tau \mid \forall s{\in}[\tau]_\approx.\,\forall u{\in}U.\,\forall(P,N){\in}\mathcal{N}(s).\,(P{\subseteq}\mathscr{A}_u \Rightarrow C_u^{P,N\cap\mathscr{A}_u})\}$$

where:

$$C^{P,N}_{\mathsf{basic}} \stackrel{\text{def}}{=} \bigcap_{b\in P}\mathbb{B}(b) \subseteq \bigcup_{b\in N\cap\mathscr{A}_{\mathsf{basic}}}\mathbb{B}(b)$$

$$C^{P,N}_{\mathsf{prod}} \stackrel{\text{def}}{=} \forall N'{\subseteq}N. \begin{cases} \mathcal{N}(\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1) \in \mathscr{S} \\ \vee \\ \mathcal{N}(\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2) \in \mathscr{S} \end{cases}$$

$$C^{P,N}_{\mathsf{fun}} \stackrel{\text{def}}{=} \exists t_0 \to s_0 \in N\,.\,\forall P' \subseteq P\,. \begin{cases} \mathcal{N}(t_0 \wedge \bigwedge_{t\to s\in P'} \neg t) \in \mathscr{S} \\ \vee \\ \begin{cases} P \neq P' \\ \wedge \\ \mathcal{N}((\neg s_0) \wedge \bigwedge_{t\to s\in P\setminus P'} s) \in \mathscr{S} \end{cases} \end{cases}$$

We say that $\mathscr{S}$ is a simulation *if:*

$$\mathscr{S} \subseteq \mathbb{E}\mathscr{S}$$

The notion of simulation is at the basis of our subtyping algorithm. In what follows we show that simulations soundly and completely characterize the set of empty types of a well-founded convex model. More precisely, we show that every type in a simulation is empty (soundness) and that the set of all empty types is a simulation (completeness), actually, the largest simulation.

Lemma 3.20. *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation and t a type. If $\forall\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\,.\,[\![t]\!]\eta = \varnothing$, then $\forall\theta : \mathcal{V} \to \mathcal{T}\,.\,\forall\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\,.\,[\![t\theta]\!]\eta = \varnothing$*

Proof. Suppose there exists $\theta : \mathcal{V} \to \mathcal{T}$ and $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$ such that $[\![t\theta]\!]\eta \neq \varnothing$. Then we consider the semantic assignment η' such that

$$\forall\alpha \in \mathsf{var}(t)\,.\,\eta'(\alpha) = [\![\theta(\alpha)]\!]\eta$$

Thus, we have $[\![t]\!]\eta' = [\![t\theta]\!]\eta \neq \varnothing$. It contradicts that $\forall\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\,.\,[\![t]\!]\eta = \varnothing$. Therefore, the result follows. □

Lemma 3.20 shows that if a type is empty, then all its syntactic instances are empty. In particular if a type is empty we can rename all its type variables without changing any property. So when working with empty types or, equivalently, with subtyping relations, types can be considered equivalent modulo α-renaming (*ie*, the renaming of type variables).

The first result we prove is that every simulation contains only empty types.

Theorem 3.21 (Soundness). *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a convex structural interpretation and $\mathscr{S}$ a simulation. Then for all $\tau \in \mathscr{S}$, we have $\forall\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\,.\,[\![\tau]\!]\eta = \varnothing$*

Proof. Consider a type $\tau \in \mathscr{S}$. Then $\forall\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\,.\,[\![\tau]\!]\eta = \varnothing$ holds, if and only if

$$\forall d \in \mathcal{D}\,.\,\forall\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\,.\,d \notin [\![\tau]\!]\eta$$

Let us take $d \in \mathcal{D}$ and $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$. Since the interpretation is structural we can prove this property by induction on d. Since $\mathscr{S}$ is a simulation, we also have $\tau \in \mathbb{E}\mathscr{S}$, that is:

$$\forall s{\in}[\tau]_\approx\,.\,\forall u{\in}U\,.\,\forall(P,N){\in}\mathcal{N}(s).\,(P{\subseteq}\mathscr{A}_u \Rightarrow C_u^{P,N\cap\mathscr{A}_u}) \quad (18)$$

where the conditions $C_u^{P,N}$ are the same as those in Definition 3.19. The result then will follow from proving a statement stronger than the one of the lemma, that is, if $\tau\in\mathbb{E}\mathscr{S}$, then $\forall d\in\mathcal{D}.\forall\eta\in\mathscr{P}(\mathcal{D})^{\mathcal{V}}.d\notin[\![\tau]\!]\eta$. Or, equivalently:

$$\begin{array}{l}\forall s\in[\tau]_{\approx}\ .\ \forall u\in U'\ .\ \forall(P,N)\in\mathcal{N}(s)\ .\\ \quad(P\subseteq\mathscr{A}_u\Rightarrow C_u^{P,N\cap\mathscr{A}_u})\Rightarrow d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N}[\![a]\!]\eta\end{array}\tag{19}$$

Let us take $s\in[\tau]_{\approx}$, $(P,N)\in\mathcal{N}(s)$ and u be the kind of d. Let us consider the possible cases for the an atom $a\in P$. Consider first $a\notin\mathcal{V}$: if $a\in\mathscr{A}\setminus\mathscr{A}_u$, then clearly $d\notin[\![a]\!]\eta$. If $a\in\mathcal{V}$ then we know (by Lemma 3.11 if $u=\mathsf{basic}$, Lemma 3.13 if $u=\mathsf{prod}$, and Lemma 3.14 if $u=\mathsf{fun}$) that (19) holds if and only if it holds for an $s\in[\tau]_{\approx}$ in which the variable $a\notin P$. As a consequence if we can prove the results $P\subseteq\mathscr{A}_u$ then (19) holds.

So assume that $P\subseteq\mathscr{A}_u$. Applying (18), we obtain that $C_u^{P,N\cap\mathscr{A}_u}$ holds. It remains to prove that:

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N}[\![a]\!]\eta$$

$u=\mathsf{basic}, d=c$. The condition $C_{\mathsf{basic}}^{P,N\cap\mathscr{A}_{\mathsf{basic}}}$ is:

$$\forall\eta\in\mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \bigcap_{b\in P}\mathbb{B}(b)\subseteq\bigcup_{b\in N\cap\mathscr{A}_{\mathsf{basic}}}\mathbb{B}(b)$$

As a consequence, we get:

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N\cap\mathscr{A}_{\mathsf{basic}}}[\![a]\!]\eta$$

and a fortiori:

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus(\bigcup_{a\in N\cap\mathscr{A}_{\mathsf{basic}}}[\![a]\!]\eta\cup\bigcup_{a\in N\setminus\mathscr{A}_{\mathsf{basic}}}[\![a]\!]\eta)$$

which yields the result.

$u=\mathsf{prod}, d=(d_1,d_2)$. The condition $C_u^{P,N\cap\mathscr{A}_u}$ is:

$$\forall N'\subseteq N\cap\mathscr{A}_{\mathsf{prod}}\ .\begin{cases}\mathcal{N}(\bigwedge_{t_1\times t_2\in P}t_1\wedge\bigwedge_{t_1\times t_2\in N'}\neg t_1)\in\mathscr{S}\\ \vee\\ \mathcal{N}(\bigwedge_{t_1\times t_2\in P}t_2\wedge\bigwedge_{t_1\times t_2\in N\setminus N'}\neg t_2)\in\mathscr{S}\end{cases}$$

For each N', we apply the induction hypothesis to d_1 and to d_2. We get:

$$d_1\notin[\![\bigwedge_{t_1\times t_2\in P}t_1\wedge\bigwedge_{t_1\times t_2\in N'}\neg t_1]\!]\eta\ \vee\ d_2\notin[\![\bigwedge_{t_1\times t_2\in P}t_2\wedge\bigwedge_{t_1\times t_2\in N\setminus N'}\neg t_2]\!]\eta$$

That is:

$$d\notin(\bigcap_{t_1\times t_2\in P}[\![t_1]\!]\eta\setminus\bigcup_{t_1\times t_2\in N'}[\![t_1]\!]\eta)\times(\bigcap_{t_1\times t_2\in P}[\![t_2]\!]\eta\setminus\bigcup_{t_1\times t_2\in N\setminus N'}[\![t_2]\!]\eta)$$

Since to $[\![t_1]\!]\eta\times[\![t_2]\!]\eta=[\![t_1\times t_2]\!]\eta$, then we get:

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N\cap\mathscr{A}_{\mathsf{prod}}}[\![a]\!]\eta$$

and a fortiori:

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N}[\![a]\!]\eta$$

$u=\mathsf{fun}, d=\{(d_1,d_1'),\ldots,(d_n,d_n')\}$. The condition $C_u^{P,N\cap\mathscr{A}_u}$ states that there exists $t_0\to s_0\in N$ such that, for all $P'\subseteq P$

$$\mathcal{N}(t_0\wedge\bigwedge_{t\to s\in P'}\neg t)\in\mathscr{S}\vee\begin{cases}P'\neq P\\ \wedge\\ \mathcal{N}((\neg s_0)\wedge\bigwedge_{t\to s\in P\setminus P'}s)\in\mathscr{S}\end{cases}$$

Applying the induction hypothesis to the d_i and d_i' (note that if $d_i'=\Omega$, then $d_i'\notin[\![\tau]\!]\eta$ is trivial for all τ):

$$d_i\notin[\![t_0\wedge\bigwedge_{t\to s\in P'}\neg t]\!]\eta\vee\begin{cases}P\neq P'\\ \wedge\\ d_i'\notin[\![(\neg s_0)\wedge\bigwedge_{t\to s\in P\setminus P'}s]\!]\eta\end{cases}$$

Assume that $\forall i\ .\ (d_i\in[\![t_0]\!]\eta\Rightarrow d_i'\in[\![s_0]\!]\eta)$. Then we have $d\in[\![t_0\to s_0]\!]\eta$.
Otherwise, let us consider i such that $d_i\in[\![t_0]\!]\eta$ and $d_i'\notin[\![s_0]\!]\eta$. The formula above gives for any $P'\subseteq P$:

$$(d_i\in\bigcup_{t\to s\in P'}[\![t]\!]\eta)\vee(P'\neq P\wedge d_i'\in\{\Omega\}\cup\bigcup_{t\to s\in P\setminus P'}[\![\neg s]\!]\eta)$$

Let's take $P'=\{t\to s\in P\mid d_i\notin[\![t]\!]\eta\}$. We have $d_i\notin\bigcup_{t\to s\in P'}[\![t]\!]\eta$, and thus $P'\neq P$ and $d_i'\in\{\Omega\}\cup\bigcup_{t\to s\in P\setminus P'}[\![\neg s]\!]\eta$. We can thus find $t\to s\in P\setminus P'$ such that $d_i'\notin[\![s]\!]\eta$,and because $t\to s\notin P'$, we also have $d_i\in[\![t]\!]\eta$. We have thus proved that $d\notin[\![t\to s]\!]\eta$ for some $t\to s\in P$.
In both cases, we get:

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N\cap\mathscr{A}_{\mathsf{fun}}}[\![a]\!]\eta$$

and a fortiori

$$d\notin\bigcap_{a\in P}[\![a]\!]\eta\setminus\bigcup_{a\in N}[\![a]\!]\eta$$

□

The intuition of the simulation is that if we consider the statements of Lemmas 3.15 and 3.16 as if they were rewriting rules (from right to left), then $\mathbb{E}\mathscr{S}$ contains all the types that we can deduce to be empty in one step reduction when we suppose that the types in $\mathscr{S}$ are empty. A simulation is thus a set that is already saturated with respect to such a rewriting. In particular, if we consider the statements of Lemmas 3.15 and 3.16 as inference rules for determining when a type is equal to $\mathbb{0}$, then $\mathbb{E}\mathscr{S}$ is the set of immediate consequences of $\mathscr{S}$, and a simulation is a *self-justifying* set, that is a co-inductive proof of the fact that all its elements are equal to $\mathbb{0}$.

Completeness derives straightforwardly from the construction of simulation and the lemmas we proved about it.

Theorem 3.22. *Let $[\![\_]\!]:\mathcal{T}\to\mathscr{P}(\mathcal{D})^{\mathcal{V}}\to\mathscr{P}(\mathcal{D})$ be a convex set-theoretic interpretation. We define a set of normal forms $\mathscr{S}$ by:*

$$\mathscr{S}\stackrel{\text{def}}{=}\{\tau\mid\forall\eta\in\mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ [\![\tau]\!]\eta=\varnothing\}$$

Then:

$$\mathbb{E}\mathscr{S}=\{\tau\mid\forall\eta\in\mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \mathbb{E}(\tau)\eta=\varnothing\}$$

Proof. Immediate consequence of Lemmas 3.15, 3.16 and 3.20. □

Corollary 3.23. *Let $[\![\_]\!]:\mathcal{T}\to\mathscr{P}(\mathcal{D})^{\mathcal{V}}\to\mathscr{P}(\mathcal{D})$ be a convex structural interpretation. Define as above $\mathscr{S}=\{\tau\mid\forall\eta\in\mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \mathbb{E}(\tau)\eta=\varnothing\}$. Then $[\![\_]\!]$ is a model if and only if $\mathscr{S}=\mathbb{E}\mathscr{S}$.*

This corollary has two implications: first, that the condition for a set-theoretic interpretation to be a model depends only on the subtyping relation it induces; second, that the simulation that contains all the empty types is the largest simulation. In particular this second implication entails the following corollary.

Corollary 3.24 (Completeness). *Let $[\![\_]\!]:\mathcal{T}\to\mathscr{P}(\mathcal{D})^{\mathcal{V}}\to\mathscr{P}(\mathcal{D})$ be a well-formed convex model and s and t two types. Then $s\leq t$ if and only if there exists a simulation $\mathscr{S}$ such that $\mathcal{N}(s\wedge\neg t)\in\mathscr{S}$.*

The corollary states that to check whether $s \leq t$ we have to check whether there exists a simulation that contains the normal form, denoted by τ_0, of $s \wedge \neg t$. Thus a simple subtyping algorithm is to use Definition 3.19 as a set of saturation rules: we start from τ_0 and try to saturate it. At each step of saturation we add just the normal forms in which we eliminated top-level variables according to Lemmas 3.10, 3.11, 3.13, and 3.14. Because of the presence of "or" in the definition, the algorithm follows different branches until it reaches a simulation (in which case it stops with success) or it adds a non-empty type (in which case the whole branch is abandoned).

All results we stated so far have never used the regularity of types. The theory holds also for non regular types and the algorithm described above is a sound and complete procedure to check their inclusion. The only result that needs regularity is decidability.

3.5 Decidability

As anticipated in the related work section, the subtyping relation on polymorphic regular types is decidable in EXPTIME. [8] We prove decidability but the result on complexity is due to Gesbert *et al.* [15] who, as we already hinted, gave a linear encoding of the relation presented here in their variant of the μ-calculus, for which they have an EXPTIME solver [14], thus obtaining a subtyping decision algorithm that is EXPTIME [9] (in doing so they also spotted a subtle error in the original definition of our subtyping algorithm).

To prove decidability we just prove that our algorithm terminates. We do so by first showing that it terminates on finite trees (which is the crux of the problem since the only potential source of loop are the substitutions of free variables performed in ***Step 5***) and then show that when we switch to regular trees the algorithm always ends up on memoized terms (see Appendix A).

More precisely, the substitutions of free variables performed in step 5 may be problematic since not only they increase the size of the terms but they also introduce fresh variables that could later surface at top level thus triggering an infinite suite of substitutions. This is not the case for finite tree types for which the procedure always terminates (see Appendix A, in particular Lemma A.2). The intuition underlying this result is that if we follow the evolution of an occurrence of a variable in the algorithm, then this occurrence can generate only finite types. Furthermore, instantiation adds only a finite number of distinct new atoms (*ie*, $\gamma_1 \times \gamma_2$ and $\gamma_1 \rightarrow \gamma_2$), and the equivalence of empty types modulo α-renaming implies that we need only a finite number of fresh variables. All this suggests that the algorithms should terminate for regular types, as well: in passing from finite trees to finitely many distinct subtrees we do not touch at the source of possible divergence. And indeed this is the case since we can show that the algorithm does not touch modify the form of the types.

3.6 Convex Models

The last step of our development is to prove that there exists at least one set-theoretical model that is convex. From a practical point of view this step is not necessary since one can always take the work we did so far as a syntactic definition of the subtyping relation. However, the mathematical support makes our theory general and applicable to several different domains (*eg*, Gesbert *et al*'s starting point and early attempts relied on this result), so finding a model is not done just for the sake of the theory. As it turns out, there actually exist a lot of convex models since every model for ground types with infinite denotations is convex. So to define a convex model it just suffices to take any model defined in [11] and straightforwardly modify the interpretation of basic and singleton types (more generally, of all indivisible types[10]) so they have infinite denotations.

Definition 3.25 (Infinite support). *A model* $(\mathcal{D}, [\![]\!])$ *is* with infinite support *if for every ground type* t *and assignment* η, *if* $[\![t]\!]\eta \neq \varnothing$, *then* $[\![t]\!]\eta$ *is an infinite set.*

What we want to prove then is the following theorem.

Theorem 3.26. *Every well-founded model with infinite support is convex.*

The proof of this theorem is quite technical—it is the proof that required us most effort—and proceeds in three logical steps. First, we prove that the theorem holds when all types at issue do not contain any product or arrow type. In other words, we prove that equation (8) holds for $[\![.]\!]$ with infinite support and where all t_i's are Boolean combinations of type variables and basic types. This is the key step in which we use the hypothesis of infinite support, since in the proof—done by contradiction—we need to pick an unbounded number of elements from our basic types in order to build a counterexample that proves the result. Second, we extend the previous proof to any type t_i that contains finitely many applications of the product constructor. In other terms, we prove the result for any (possibly infinite) type, provided that recursion traverses just arrow types, but not product types. As in the first case, the proof builds some particular elements of the domain. In the presence of type constructors the elements are built inductively. This is possible since products are not recursive, while for arrows it is always possible to pick a fresh appropriate element that resides in that arrow since every arrow type contains the (indivisible) closed type $\mathbb{1} \rightarrow \mathbb{0}$. Third, and last, we use the previous result and the well-foundedness of the model to show that the result holds for all types, that is, also for types in which recursion traverses a product type. More precisely, we prove that if we assume that the result does not hold for some infinite product type then it is possible to build a finite product type (actually, a finite expansion of the infinite type) that disproves equation (8), contradicting what is stated in our second step. Well-foundedness of the model allows us to build this finite type by induction on the elements denoted by the infinite one.

More precisely, we proceed as follows.

Definition 3.27 (Positive and negative occurrences). *Let* $t \in \mathcal{T}$ *and* t' *be a tree that occurs in* t. *An occurrence of* t' *in* t *is a* negative occurrence *of* t *if on the path going from the root of the occurrence to the root of* t *there is an odd number of* $\neg$ *nodes. It is a* positive occurrence *otherwise.*

For instance if $t = \neg(\alpha \times \neg\beta)$ then β is a positive occurrence of t while α and $\alpha \times \neg\beta$ are negative ones.

Definition 3.28. *We use* $\mathcal{T}^{\mathsf{fp}}$ *to denote the set of* types with finite products*, that is, the set of all types in which every infinite branch contains a finite number of occurrences of the* $\times$ *constructor.*

The first two steps of our proof are proved simultaneously in the following lemma.

[8] Strictly speaking we cannot speak of *the* subtyping relation in general, but just of the subtyping relation induced by a particular model. Here we intend the subtyping relation for the model used in the proof of Corollary 3.32, that is, of the universal model of [11] with a set of basic types whose containment relation is decidable in EXPTIME. This coincides with the relation studied in [15].

[9] This is also a lower bound for the complexity since the subtyping problem for ground regular types (without arrows) is known to be EXPTIME-complete.

[10] Our system has a very peculiar indivisible type: $\mathbb{1} \rightarrow \mathbb{0}$, the type of the functions that diverge on all arguments. This can be handled by adding a fixed infinite set of fresh elements of the domain to the interpretation of every arrow type (*cf.* the proof of Corollary 3.32).

Lemma 3.29. *Let $(\mathcal{D}, [\![\_]\!])$ be a well-founded model with infinite support, and $t_i \in \mathcal{T}^{\mathsf{fp}}$ for $i \in [1..n]$. Then*

$$\forall\eta\ .\ [\![t_1]\!]\eta = \varnothing \vee \ldots \vee [\![t_n]\!]\eta = \varnothing \Leftrightarrow$$
$$\forall\eta\ .\ [\![t_1]\!]\eta = \varnothing \vee \ldots \vee \forall\eta\ .\ [\![t_n]\!]\eta = \varnothing$$

Proof. The $\Leftarrow$ direction is trivial. For the other direction we proceed as follows. If at most one type is not ground, then the result is trivial. If $\mathsf{var}(t_i) \cap \mathsf{var}(t_j) = \varnothing$ for any two types t_i and t_j, that is, the sets of variables occurring in the types are pairwise disjoint, then the result follows since there is no correlation between the different interpretations. Assume that $\bigcup_{1 \le i < j \le n}(\mathsf{var}(t_i) \cap \mathsf{var}(t_j)) \neq \varnothing$. For convenience, we write CV for the above set of common variables. Suppose by contradiction that

$$\forall\eta\ .\ [\![t_1]\!]\eta = \varnothing \vee \ldots \vee \forall\eta\ .\ [\![t_n]\!]\eta = \varnothing \qquad (20)$$

does not hold. Then for each $i \in [1..n]$ there exists η_i such that $[\![t_i]\!]\eta_i \neq \varnothing$. If under the hypothesis that (20) does not hold we can find another assignment η' such that $[\![t_i]\!]\eta' \neq \varnothing$ for every $i \in [1..n]$, then this contradicts the hypothesis of the lemma and the result follows. We proceed by a case analysis on all possible combinations of the t_i's: under the hypothesis of existence of η_i such that $[\![t_i]\!]\eta' \neq \varnothing$, we show how to build the η' at issue.

Case 1: Each t_i is a single normal form, that is, $t_i \in \mathscr{P}_f(\mathscr{A} \cup \mathcal{V}) \times \mathscr{P}_f(\mathscr{A} \cup \mathcal{V})$. Recall that we want to show that it is possible to construct an assignment of the common variables such that all the t_i's have non empty denotation. To do that we build the assignment for each variable step by step, by considering one t_i at a time, and adding to the interpretation of the common variables one element after one element. In order to produce for a given type t_i an assignment that does not interfere with the interpretation defined for a different t_j, we keep track in a set s_0 of the elements of the domain $\mathcal{D}$ we use during the construction. Since we start with an empty s_0 and we add to it an element at a time, then at each step s_0 will be finite and, thanks to the property of infinite support, it will always be possible to choose some element of the domain that we need to produce our assignment and that is not in s_0. More precisely, we construct a set s_0 such that $s_0 \cap [\![t'_i]\!]\eta_i \neq \varnothing$ for each $i \in [1..n]$ where t'_i is obtained from t_i by eliminating the top-level variables. Meanwhile, considering each variable $\alpha \in \bigcup_{i \in \{1,\ldots,n\}} \mathsf{var}(t_i)$ (clearly including CV), we also construct a set s_α which is initialized as an empty set and that at the end of this construction is used to define $\eta'(\alpha)$.

Subcase 1: $\mathcal{N}(t_i) = \bigwedge_{j \in J_1} b^1_j \wedge \bigwedge_{j \in J_2} \neg b^2_j \wedge \bigwedge_{j \in J_3} \alpha^1_j \wedge \bigwedge_{j \in J_4} \neg\alpha^2_j$, where J_i's are finite sets. The construction is as follows:

If $\exists d\ .\ d \in [\![t_i]\!]\eta_i \wedge d \notin s_0$, then set $s_0 = s_0 \cup \{d\}$. For each variable $\alpha \in \mathsf{var}(t_i)$, if α is positive then $s_\alpha = s_\alpha \cup \{d\}$. If such a d does not exist, then t_i is not ground, since $[\![t_i]\!]\eta_i \subseteq s_0$ and s_0 is finite. As $\mathcal{N}(t_i) = \bigwedge_{j \in J_1} b^1_j \wedge \bigwedge_{j \in J_2} \neg b^2_j \wedge \bigwedge_{j \in J_3} \alpha^1_j \wedge \bigwedge_{j \in J_4} \neg\alpha^2_j$, then either $J_1 \cup J_2 = \varnothing$ and then we chose any element d' such that $d' \notin s_0$, or $[\![\bigwedge_{j \in J_1} b^1_j \wedge \bigwedge_{j \in J_2} \neg b^2_j]\!]\eta_i$ is non empty (because $[\![t'_i]\!]\eta_i \neq \varnothing$) and infinite (since the type at issue is closed and $[\![.]\!]$ is with infinite support), and then $\exists d'\ .\ d' \in [\![\bigwedge_{j \in J_1} b^1_j \wedge \bigwedge_{j \in J_2} \neg b^2_j]\!]\eta_i \wedge d' \notin s_0$. In both cases we set $s_0 = s_0 \cup \{d'\}$, and for each variable $\alpha \in \mathsf{var}(t_i)$, if α is positive we set $s_\alpha = s_\alpha \cup \{d\}$.

Subcase 2: $\mathcal{N}(t_i) = \bigwedge_{j \in J_1}(t^1_j \times t^2_j) \wedge \bigwedge_{j \in J_2} \neg(t^3_j \times t^4_j) \wedge \bigwedge_{j \in J_3} \alpha^1_j \wedge \bigwedge_{j \in J_4} \neg\alpha^2_j$, where J_i's are finite sets. If $|J_1| = |J_2| = 0$, then we are in the case of Subcase 1. If $|J_1| = 0$ and $|J_2| \neq 0$, then we can do the construction as Subcase 1 since $\mathscr{C} \subseteq [\![\bigwedge_{j \in J_2} \neg(t^3_j \times t^4_j)]\!]\eta_i$. Suppose then that $|J_1| > 0$: since we have

$$\bigwedge_{j \in J_1}(t^1_j \times t^2_j) = (\bigwedge_{j \in J_1} t^1_j \times \bigwedge_{j \in J_1} t^2_j)$$

then without loss of generality we can assume that $|J_1| = 1$, that is there is a single toplevel non negated product type. So we are considering the specific case for

$$\mathcal{N}(t_i) = (t'_1 \times t'_2) \wedge \bigwedge_{j \in J_3} \alpha^1_j \wedge \bigwedge_{j \in J_4} \neg\alpha^2_j.$$

What we do next is to build a particular element of the domain by exploring $(t'_1 \times t'_2)$ in a top-down way and stopping when we arrive to a basic type, or a variable, or *an arrow type*. So even though $(t'_1 \times t'_2)$ may be infinite (since it may contain an arrow type of infinite depth) the exploration will always terminate (unions, negations, *and* products always are finite: in particular products are finite because by hypothesis we are considering only types in $\mathcal{T}^{\mathsf{fp}}$). It can then be defined recursively in terms of two mutually recursive explorations that return different results according to whether the exploration step has already crossed an even or an odd number of negations. So let t_1 be a type different from $\mathbb{0}$ and t_2 a type different from $\mathbb{1}$, we define the explore_pos(t_1) and explore_neg(t_2) procedures that, intuitively, explore the syntax tree for positive and negative occurrences, respectively, and which are defined as follows:

explore_pos(t) case t of:

1. $t = t_1 \times t_2$. Let d_i be the result of explore_pos(t_i) (for $i = 1, 2$): since t is not empty so must t_1 and t_2 be; we add both d_1 and d_2 to s_0 and return $d = (d_1, d_2)$.
2. $t = t_1 \to t_2$: we can choose any element $d \in \mathbb{1} \to \mathbb{0}$ (we need not to consider t_1, t_2) and return it.
3. $t = \mathbb{0}$: impossible.
4. $t = \mathbb{1}$: return any element $d \notin s_0$.
5. $t = b$: we can choose any element $d \in [\![b]\!]\eta_i$ and $d \notin s_0$ and return it.
6. $t = \alpha$: we can choose any element $d \notin s_0$, set $s_\alpha = s_\alpha \cup \{d\}$ and return d.
7. $t = t_1 \vee t_2$: one of the two types is not empty. If it is t_1, then call explore_pos(t_1). It yields $d_1 \notin s_0$ and we return it. Otherwise we call explore_pos(t_2) and return its result.
8. $t = t_1 \wedge t_2$, then put it in disjunctive normal form. Since it is not empty, then then one of its single normal forms must be non empty, as well: repeat for this non empty single normal form the construction of the corresponding subcase in this proof and return the d constructed by it.
9. $t = \neg t'$, then we call explore_neg(t') add its result to s_0 and return it.

explore_neg(t), case t of:

1. $t = t_1 \times t_2$: we can choose any element $d \in \mathscr{C}$ and $d \notin s_0$ and return it.
2. $t = t_1 \to t_2$: we can choose any element $d \in \mathscr{C}$ and $d \notin s_0$ and return it.
3. $t = \mathbb{0}$: return any element $d \notin s_0$.
4. $t = \mathbb{1}$: impossible.
5. $t = b$: we can choose any element $d \notin [\![b]\!]\eta_i$ and $d \notin s_0$ and return it.
6. $t = \alpha$: we can choose any element $d \notin s_0$ (which clearly implies that $d \notin s_\alpha$), and return it.
7. $t = (t_1 \vee t_2)$: call explore_pos($\neg t_1 \wedge \neg t_2$) and return it.
8. $t = (t_1 \wedge t_2)$: since this intersection is not $\mathbb{1}$ then one of the two types is not $\mathbb{1}$. If it is t_1 then call explore_neg(t_1) and return it else return explore_neg(t_2).
9. $t = \neg t'$, then we call explore_pos(t') and return it.

Let $d = \mathsf{explore\_pos}(t'_1 \times t'_2)$. Since $[\![t'_1 \times t'_2]\!]\eta_i \neq \varnothing$, then the call is well defined. Then set $s_0 = s_0 \cup \{d\}$ and $s_{\alpha^1_j} = s_{\alpha^1_j} \cup \{d\}$ for all $j \in J_3$.

Finally there is the case in which also $|J_2| > 0$, that is, there exists at least one toplevel negative product type. Since we have

$$(t_1 \times t_2) \wedge \neg(t_3 \times t_4) = (t_1 \backslash t_3 \times t_2) \vee (t_1 \times t_2 \backslash t_4)$$

then we can do the construction as above either for $(t_1 \setminus t_3 \times t_2)$ or $(t_1 \times t_2 \setminus t_4)$: multiple negative product types are treated in the same way.

Subcase 3: $\mathcal{N}(t_i) = \bigwedge_{j\in J_1}(t^1_j \to t^2_j) \wedge \bigwedge_{j\in J_2} \neg(t^3_j \to t^4_j) \wedge \bigwedge_{j\in J_3} \alpha^1_j \wedge \bigwedge_{j\in J_4} \neg\alpha^2_j$, where J_i's are finite sets. If $|J_1| = |J_2| = 0$, then we are in the case of Subcase 1. If $|J_1| = 0$ and $|J_2| \neq 0$, then we can do the construction as Subcase 1 since $\mathscr{C} \subseteq [\![\bigwedge_{j\in J_2} \neg(t^3_j \to t^4_j)]\!]\eta_i$. Therefore let us suppose that $|J_1| \neq 0$. The remaining two cases, that is, $|J_2| = 0$ and $|J_2| \neq 0$, deserve to be treated separately:

$|J_2| = 0$ In this case we have at toplevel an intersection of arrows and no negated arrow. Notice that for all $j \in J_1$ we have $\mathbb{1}{\to}\mathbb{0} \leq t^1_j{\to}t^2_j$, therefore we deduce that $\mathbb{1}{\to}\mathbb{0} \leq \bigwedge_{j\in J_1}(t^1_j \to t^2_j)$. Since $\mathbb{1}{\to}\mathbb{0}$ is a closed type (actually, a indivisible one) and $[\![.]\!]$ is with infinite support, then the denotation of $\mathbb{1}{\to}\mathbb{0}$ contains infinitely many elements. Since s_0 is finite, then it is possible to choose a d in the denotation of $\mathbb{1}{\to}\mathbb{0}$ such that $d \notin s_0$. Once we have chosen such a d, we proceed as before, namely, we set $s_0 = s_0 \cup \{d\}$ and similarly add d to s_α for every variable α occurring positively at the top-level of t_i (*ie*, for all α^1_j with $j \in J_3$).

$|J_2| \neq 0$ This case cannot be solved as for $|J_2| = 0$, insofar as we can no longer find a closed type that is contained in $\bigwedge_{j\in J_1}(t^1_j \to t^2_j) \wedge \bigwedge_{j\in J_2} \neg(t^3_j \to t^4_j)$: since we have at least one negated arrow type, then $\mathbb{1} \to \mathbb{0}$ is no longer contained in the intersection. The only solution is then to build a particular element in this intersection in the same way we did for product types in Subcase 2. Unfortunately, contrary to the case of product types, we cannot work directly on the interpretation function $[\![.]\!]$ since we do not know its definition on arrow types. However, since we are in a model, we know its behavior with respect to its associated extensional interpretation $\mathbb{E}_{[\![\,]\!]}$, namely, that for every assignment η and type t it holds $[\![t]\!]\eta = \varnothing \iff \mathbb{E}(t)\eta = \varnothing$. Since we supposed that there exist n assignments η_i such that $[\![t_i]\!]\eta_i \neq \varnothing$ (for $i \in [1..n]$), then the model condition implies that for these same assignments $\mathbb{E}(t_i)\eta_i \neq \varnothing$. If from this we can prove that there exists an assignment η' such that for all $i \in [1..n]$, $\mathbb{E}(t_i)\eta' \neq \varnothing$, then by the model condition again we can deduce that for all $i \in [1..n]$, $[\![t_i]\!]\eta' \neq \varnothing$, that is our thesis.[11]

Consider $\bigwedge_{j\in J_1}(t^1_j \to t^2_j) \wedge \bigwedge_{j\in J_2} \neg(t^3_j \to t^4_j)$. By hypothesis we have $\mathbb{E}(\bigwedge_{j\in J_1}(t^1_j \to t^2_j) \wedge \bigwedge_{j\in J_2} \neg(t^3_j \to t^4_j))\eta_i \neq \varnothing$. By definition of $\mathbb{E}$ this is equivalent to

$$\bigcap_{j\in J_1}([\![t^1_j]\!]\eta_i \to [\![t^2_j]\!]\eta_i) \cap \bigcap_{j\in J_2} \neg([\![t^3_j]\!]\eta_i \to [\![t^4_j]\!]\eta_i) \neq \varnothing,$$

or equivalently,

$$\bigcap_{j\in J_1} \mathscr{P}(\overline{[\![t^1_j]\!]\eta_i \times \overline{[\![t^2_j]\!]\eta_i}}) \cap \bigcap_{j\in J_2} \neg\mathscr{P}(\overline{[\![t^3_j]\!]\eta_i \times \overline{[\![t^4_j]\!]\eta_i}}) \neq \varnothing$$

We want to construct an assignment η' and a set of pairs such that this set of pairs is included in the intersection above. We then use this set of pairs to define our assignment η'. According to Lemma 6.8 in [11], the intersection above is not empty if and only if

$$\forall j_2 \in J_2 \,.\, \exists J' \subseteq J_1 \,.$$
$$\begin{cases} [\![t^3_{j_2} \setminus \bigvee_{j_1\in J'} t^1_{j_1}]\!]\eta_i \neq \varnothing & \text{if } J_1 = J' \\ \begin{cases} [\![t^3_{j_2} \setminus \bigvee_{j_1\in J'} t^1_{j_1}]\!]\eta_i \neq \varnothing \\ \wedge \\ [\![\bigwedge_{j_1\in J_1\setminus J'} t^2_{j_1} \setminus t^4_{j_2}]\!]\eta_i \neq \varnothing \end{cases} & \text{otherwise} \end{cases}$$

Therefore, consider each $j_2 \in J_2$ and let J^{j_2} denote a subset $J' \subseteq J_1$ for which the property above holds. Then we proceed as we did in the Subcase 2 and use $\mathsf{explore\_pos}$ to build *two* elements $d^1_{j_2}$ and $d^2_{j_2}$. More precisely, if $J^{j_2} \neq J_1$ then we set $d^1_{j_2} = \mathsf{explore\_pos}(t^3_{j_2} \setminus \bigvee_{j_1\in J^{j_2}} t^1_{j_1})$ and $d^2_{j_2} = \mathsf{explore\_pos}(\bigwedge_{j_1\in J_1\setminus J^{j_2}} t^2_{j_1} \setminus t^4_{j_2})$; if $J^{j_2} = J_1$, then we set $d^1_{j_2} = \mathsf{explore\_pos}(t^3_{j_2} \setminus \bigvee_{j_1\in J_1} t^1_{j_1})$ and $d^2_{j_2} = \mathsf{explore\_pos}(\neg t^4_{j_2})$ (actually, any type is ok provided that we pick an element not in s_0). We add $d^1_{j_2}, d^2_{j_2}$, and $(d^1_{j_2}, d^2_{j_2})$ to s_0. Now consider the various pairs of the form $(d^1_{j_2}, d^2_{j_2})$ for $j_2 \in J_2$. Since we chose $d^1_{j_2} \notin [\![\bigvee_{j_1\in J^{j_2}} t^1_{j_1}]\!]\eta_i$, then $(d^1_{j_2}, d^2_{j_2}) \in [\![t^1_j \to t^2_j]\!]$ for all $j \in J_1$, and therefore it belongs to the intersection $\bigcap_{j\in J_1}[\![t^1_j \to t^2_j]\!]\eta_i$. Furthermore, by construction each $(d^1_{j_2}, d^2_{j_2}) \notin [\![t^3_{j_2} \to t^4_{j_2}]\!]\eta_i$. Therefore the set of pairs $\{(d^1_{j_2}, d^2_{j_2}) \mid j_2 \in J_2\}$ is the element we were looking for: we add $\{(d^1_{j_2}, d^2_{j_2}) \mid j_2 \in J_2\}$ to s_0 and to each $s_{\alpha^1_j}$ for $j \in J_3$.

Subcase 4: The previous subcases cover all the case in which all the literals of the single normal form at issue are on the same constructor (all basic or product or arrow types). So the only remaining subcase is the one in which there are literals with different constructors. This is quite straightforward because it is always possible to reduce the problem to one of the previous cases. More precisely

1. The case in which there are two positive literals with different constructors is impossible since the type would be empty (*eg* the intersection of a basic and a product type is always empty), contradicting our hypothesis.
2. Suppose t_i contains some positive literals all on the same constructor. Then we can erase all the negative literals with a different constructor since they contain all the positive ones, thus reducing the problem to one of the previous subcases.
3. Suppose that t_i contains no positive literal on any constructor, that is it is formed only by negated literals on some constructors. Since the type is not empty, then either the union of all negated basic types does not cover $\mathscr{C}$, or the union of all negated product types does not cover $\mathbb{1} \times \mathbb{1}$, or the union of all negated arrow types does not cover $\mathbb{0} \to \mathbb{1}$. In the first case take as d any element of $\mathscr{C}$ that is neither in s_0 nor in the union of all negated basic types. In the second case, keep just the negated product types, intersect them with $\mathbb{1} \times \mathbb{1}$ and proceed as in Subcase 2. In the third case keep just the negated ar-

[11] As an aside, notice we could have used this technique also in other cases and by the very definition of $\mathbb{E}$ the proof would not have changed (apart from an initial reference to $\mathbb{E}$ at the beginning of each subcase as the one preceding this footnote). Actually, strictly speaking, we already silently used this technique in the case of products since the hypothesis of well-foundedness of model does not state that $[\![t_1 \times t_2]\!]\eta$ is equal to $[\![t_1]\!]\eta \times [\![t_2]\!]\eta$ (an assumption we implicitly did all the proof long) but just that induces the same subtyping relation as a model in which that equality holds. We preferred not to further complicate the presentation of that case.

row types, intersect them with $\mathbb{0} \to \mathbb{1}$ and proceed as in Subcase 3.

At the end of this construction we define a new semantic assignment η' as follows $\eta' = \{{}^{s_\alpha}\!/\!_\alpha, \ldots\}$ for $\alpha \in \bigcup_{i\in\{1,\ldots,n\}} \mathsf{var}(t_i)$. By construction of η' we have $[\![t_i]\!]\eta' \neq \varnothing$ for each $i \in [1..n]$, which contradicts the premise.

Case 2: There exists $i \in \{1, \ldots, n\}$ such that $t_i = t_i^1 \vee \ldots \vee t_i^m$ while t_j is a single normal form for all $j \neq i$.
Form Definition 3.2, we have

$$[\![t_i]\!]\eta = \varnothing \Leftrightarrow [\![t_i^1]\!]\eta = \varnothing \wedge \ldots \wedge [\![t_i^m]\!]\eta = \varnothing$$

Since

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_1]\!]\eta = \varnothing \vee \ldots \vee [\![t_i]\!]\eta = \varnothing \vee \ldots \vee [\![t_n]\!]\eta = \varnothing$$

Then consider each t_i^j, we have

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_1]\!]\eta = \varnothing \vee \ldots \vee [\![t_i^j]\!]\eta = \varnothing \vee \ldots \vee [\![t_n]\!]\eta = \varnothing$$

By Case 1, we have

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_1]\!]\eta = \varnothing \vee \ldots \vee \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_i^j]\!]\eta = \varnothing \vee \ldots \vee \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_n]\!]\eta = \varnothing$$

If there exists one type t_k such that $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_k]\!]\eta = \varnothing$ holds, where $k \in \{1, \ldots, n\} \setminus \{i\}$, then the result follows. Otherwise, we have

$$\begin{aligned} &\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_i^1]\!]\eta = \varnothing \wedge \ldots \wedge \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_i^m]\!]\eta = \varnothing \\ \Leftrightarrow &\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_i^1]\!]\eta = \varnothing \wedge \ldots \wedge [\![t_i^m]\!]\eta = \varnothing \\ \Leftrightarrow &\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . [\![t_i]\!]\eta = \varnothing \end{aligned}$$

Therefore the result follows.

Other cases: Similarly to Case 2. We decompose one of the types (assume t_1), then by Case 2, either one of the other types is empty, or all the decompositions of t_1 are empty, then t_1 is empty. ☐

Finally, it just remains to prove Theorem 3.26, that is to say, that Lemma 3.29 above holds also for t_i's with recursive products. This result requires the following preliminary lemma.

Lemma 3.30. *Let $[\![\_]\!]$ be a well-founded model with infinite support and t a type (which may thus contain infinite product types). If there exists a value d and an assignment $\overline{\eta}$ such that $d \in [\![t]\!]\overline{\eta}$, then there exists a type $t^{\mathsf{fp}} \in \mathcal{T}^{\mathsf{fp}}$ such that $d \in [\![t^{\mathsf{fp}}]\!]\overline{\eta}$ and for all assignment η if $[\![t]\!]\eta = \varnothing$, then $[\![t^{\mathsf{fp}}]\!]\eta = \varnothing$.*

Proof. Since the model is well founded then by Definition 3.7 we can use a well-founded preorder $\blacktriangleright$ on the elements d of the domain $\mathcal{D}$. Furthermore, since our types are regular then there are just finitely many distinct subtrees of t that are product types. So we proceed by induction on $\blacktriangleright$ and the number n of distinct subtrees of t of the form $t_1 \times t_2$ that do not belong to $\mathcal{T}^{\mathsf{fp}}$. If $n = 0$, then t already belongs to $\mathcal{T}^{\mathsf{fp}}$. Suppose that $t = t_1 \times t_2 \notin \mathcal{T}^{\mathsf{fp}}$. Then $d = (d_1, d_2)$. By induction hypothesis on d_1, d_2, there exist $t_1^{\mathsf{fp}}, t_2^{\mathsf{fp}} \in \mathcal{T}^{\mathsf{fp}}$ such that $d_i \in [\![t_i^{\mathsf{fp}}]\!]\overline{\eta}$ and for all η, $[\![t_i]\!]\eta = \varnothing \Rightarrow [\![t_i^{\mathsf{fp}}]\!]\eta = \varnothing$, for $i = 1, 2$. Then we take $t^{\mathsf{fp}} = t_1^{\mathsf{fp}} \times t_2^{\mathsf{fp}}$ and the result follows. Finally if the product at issue is not at toplevel then we can choose any (recursive) product subtree in t and we have two cases. Either the product does not "participate" to the non emptiness of $[\![t]\!]\overline{\eta}$ (*eg*, it occurs in a union addendum that is empty) and then it can be replaced by any type. Or we can decompose d to arrive to a d' that corresponds to the product subtree at issue, and then apply the induction hypothesis as above. In both cases we removed one of the distinct product subtrees that did not belong to $\mathcal{T}^{\mathsf{fp}}$ and the result follows by induction on n. ☐

While the statement of the previous lemma may, at first sight, seem obscure, its meaning is rather obvious. It states that in a well-founded model (*ie*, a model in which all the values are finite) whenever a recursive (product) type contains some value, then we can find a *finite* expansion of this type that contains the same value; furthermore, if the recursive type is empty in a given assignment, then also its finite expansion is empty in that assignment. This immediately yields our final result.

Lemma 3.31. *Let $(\mathcal{D}, [\![\_]\!])$ be a well-founded model with infinite support, and t_i for $i \in [1..n]$. Then*

$$\begin{aligned} &\forall \eta . [\![t_1]\!]\eta = \varnothing \vee \ldots \vee [\![t_n]\!]\eta = \varnothing \Leftrightarrow \\ &\forall \eta . [\![t_1]\!]\eta = \varnothing \vee \ldots \vee \forall \eta . [\![t_n]\!]\eta = \varnothing \end{aligned}$$

Proof. By Lemma 3.29 we know that if $t_i \in \mathcal{T}^{\mathsf{fp}}$ for $i \in [1..n]$ then the Lemma holds. Suppose by contradiction, that the result does not hold. Then there exists η' such that $[\![t_i]\!]\eta' \neq \varnothing$ for all $i \in [1..n]$. Let $d_i \in [\![t_i]\!]\eta'$, we can apply Lemma 3.30 and for all $i \in [1..n]$ find $t_i' \in \mathcal{T}^{\mathsf{fp}}$ such that $\forall \eta . [\![t_1']\!]\eta = \varnothing \vee \ldots \vee [\![t_n']\!]\eta = \varnothing$ and $d_i \in [\![t_i']\!]\eta'$: impossible by Lemma 3.29. ☐

Corollary 3.32 (Convex model). *There exists a convex model.*

Proof. It suffices to take any model for the ground types with an infinite domain (see [9] for examples) and interpret indivisible types into infinite sets. For instance, imagine we have n basic types $b_1, \ldots, b_n$ and suppose, for simplicity, that they are pairwise disjoint. If we use the "universal model" of [11] it yields (roughly, without the modifications for Ω) the following model. $\mathcal{D} = \mathscr{C} + \mathcal{D}^2 + \mathscr{P}_f(\mathcal{D}^2)$ where $\mathscr{C} = S_0 \cup S_1 \cup \ldots \cup S_n$ with S_i are pairwise disjoint infinite sets:

$$\begin{array}{rlrl} [\![\mathbb{0}]\!]\eta = & \varnothing & [\![\mathbb{1}]\!]\eta = & \mathcal{D} \\ [\![\neg t]\!]\eta = & \mathcal{D} \setminus [\![t]\!]\eta & [\![b_i]\!]\eta = & S_i \\ [\![t_1 \vee t_2]\!]\eta = & [\![t_1]\!]\eta \cup [\![t_2]\!]\eta & [\![t_1 \times t_2]\!]\eta = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \wedge t_2]\!]\eta = & [\![t_1]\!]\eta \cap [\![t_2]\!]\eta & [\![t_1 \to t_2]\!]\eta = & \mathscr{P}_f(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) \cup S_0 \end{array}$$

Notice that all denotations of arrow types contain S_0 thus, in particular, the indivisible type $\mathbb{1} \to \mathbb{0}$, too. If the basic types are not pairwise disjoint then it suffice to take for $\mathscr{C}$ a set of S_i whose intersections correspond to those of the corresponding basic types. The only requirement is that all intersections must be infinite sets, as well. ☐

4. Conclusion

This work constitutes the first solution to the problem of defining a semantic subtyping relation for a polymorphic extension of regular tree types. This problem, despite its important practical interest and potential fallouts, has been somehow neglected by most recent research since it was considered untreatable or unfeasible. Our solution not only has immediate application to the definition of programming languages for XML, but also opens several new research directions that we briefly discuss below.

The first direction concerns the definition of extensions of the type system itself. Among the possible extensions the most interesting (and difficult) one seems to be the extension of types with explicit second order quantifications. Currently, we consider prenex polymorphism, thus quantification on types is performed only at meta-level. But since this work proved the feasibility of a semantic subtyping approach for polymorphic types, we are eager to check whether it can be further extended to impredicative second order types, by adding explicit type quantification. This would be interesting not only from a programming language perspective, but also from a logic viewpoint since it would remove some of the limitations to the introspection capabilities we pointed out in Section 2.7. This may move our type system closer to being an expressive logic for subtyping. On the model side, it would be interesting to check whether the infinite support property (Definition 3.25) is not only a sufficient but also a necessary condition for convexity. This seems likely to the extent that the result holds for the type system restricted to basic type constructors (*ie*, without products and arrows). However, this is just a weak conjecture since the proof of sufficiency heavily relies (in the case for product types) on the well-foundedness property. Therefore, there may even exist non-well-founded models (non-well-founded models exist in the ground case: see [9, 11]) that are with infinite support but not convex. Nevertheless, an equivalent characterization of convexity—whether it were infinite support or some other characterization—would provide us a different angle of attack to study the connections between convexity and parametricity (see later on).

The second direction is to explore the definition of new languages to take advantage of the new capabilities of our system. A first natural test will be to see how to add overloaded (typed by intersection types) and higher-order (typed by arrow types) functions to the language defined in [17]. This already looks as quite a challenging problem since it needs local type inference for both subtyping and instantiation,[12] and we are actively working on it. But the overall design space for a programming language that can exploit the advanced features of our types is rather large since a lot of possible variations can be considered (*eg*, the use of type variables in pattern matching) and even more features can be encoded (*eg*, as explained in Footnote 4, bounded quantification can not only be encoded via intersection types but, thanks to them, also made more general since intersections allow the programmer to specify bounds on a per-occurrence basis). While exploring this design space it will be interesting to check whether our polymorphic union types can encode advanced type features such as polymorphic variants [13] and GADTs [27].

In our opinion, the definition of convexity is the most important and promising contribution of our work especially in view of its potential implications on the study of parametricity. As a matter of fact, there are strong connections between parametricity and convexity. We have already seen that convexity removes the stuttering behavior that clashes with parametricity, as equation (5) clearly illustrates. More generally, both convexity and parametricity describe or enforce uniformity of behavior. Parametricity imposes to functions a uniform behavior on parameters typed by type variables, since the latter cannot be deconstructed. This allows Wadler to deduce "theorems for free": the uniformity imposed by parametricity (actually, imposed by the property of being definable in second order λ-calculus) dramatically restricts the choice of possible behaviors of parametric functions to a point that it is easy to deduce theorems about a function just by considering its type [25]. In a similar way convexity imposes a uniform behavior to the zeros of the semantic interpretation, which is equivalent to imposing uniformity to the subtyping relation. An example of this uniformity is given by product types: in our framework a product $(t_1 \times ... \times t_n)$ is empty (*ie*, it is empty for every possible assignment of its variables) if and only if there exists a particular t_i that is empty (for all possible assignments). We recover a property typical of closed types.

We conjecture the connection to be much deeper than described above. This can be clearly perceived by rereading the original Reynolds paper on parametricity [21] in the light of our results. Reynolds tries to characterize parametricity—or *abstraction* in Reynolds terminology—in a set-theoretic setting since, in Reynolds words, "if types denote specific subsets of a universe then their unions and intersections are well defined", which in Reynolds opinion is the very idea of abstraction. This can be rephrased as the fact that the operations for some types are well defined independently from the representation used for each type (Reynolds speaks of abstraction and representation since he sees the abstraction property as a result about change of representation). The underlying idea of parametricity according to Reynolds is that "meanings of an expression in 'related' environments will be 'related' " [21]. But as he points out few lines later "while the relation for each type variable is arbitrary, the relation for compound type expressions [*ie*, type constructors] must be induced in a specified way. We must specify how an assignment of relations to type variables is extended to type expressions" [21]. Reynolds formalizes this extension by defining a "relation semantics" for type constructors and, as Wadler brilliantly explains [25], this corresponds to regard types as relations. In particular pairs are related if their corresponding components are related and functions are related if they take related arguments into related results: there is a precise correspondence with the extensional interpretation of type constructors we gave in Definition 3.6 and, more generally, between the framework used to state parametricity and the one in our work.

Parametricity holds for terms written in the Girard/Reynolds second order typed lambda calculus (also known as pure polymorphic lambda calculus or System F). The property of being definable in the second-order typed lambda-calculus is *the* condition that harnesses expressions and forces them to behave uniformly. Convexity, instead, does not require any definability property. It *semantically* harnesses the denotations of expressions and forces them to behave uniformly. Therefore all seems to suggest that convexity may be a semantic characterization of what in Reynolds approach is the definability in the second-order typed lambda-calculus, which is a syntactic property. Or, to put it otherwise, convexity states parametricity for (or transposes it to) models rather than languages.

Although we have this strong intuition about the connection between convexity and parametricity, we do not know how to express this connection in a formal way, yet. We believe that the answer may come from the study of the calculus associated to our subtyping relation. We do not speak here of some language that can take advantage of our subtyping relation and whose design space we discussed earlier in this conclusion. What we are speaking of is every calculus whose model of values (*ie*, the model obtained by associating each type to the set of values that have that type) induces the same subtyping relation as the one devised here. Indeed, as parametricity leaves little freedom to the definition of transformations, so the semantic subtyping framework leaves little freedom to the definition of a language whose model of values induces the same subtyping relation as the relation used to type its values. If we could prove that every such language must automatically satisfy Reynolds abstraction theorem (and, even more, prove also the converse), then we would have a formal and strong connection between convexity and parametricity, the former being a purely semantic (in the sense that it does not rely on any language or calculus) characterization of the latter. But this is a long term and ambitious goal that goes well beyond the scope of the present work.

[12] To have some flavor of the problem, consider an overloaded function `even` for the domain $\mathbb{1}$ that when applied to an integer returns whether it is even or not, while it returns arguments of any other type unchanged. Its type is $(\texttt{Int} \rightarrow \texttt{Bool}) \wedge (\alpha\backslash\texttt{Int} \rightarrow \alpha\backslash\texttt{Int})$. If we apply a currified $\texttt{map} : (\beta \rightarrow \gamma) \rightarrow \beta{*} \rightarrow \gamma{*}$ to `odd`, then we want to deduce `map(even)`: $(\texttt{Int*} \rightarrow \texttt{Bool*}) \wedge ((\alpha\backslash\texttt{Int}){*} \rightarrow (\alpha\backslash\texttt{Int}){*}) \wedge ((\alpha \vee \texttt{Int}){*} \rightarrow ((\alpha\backslash\texttt{Int}) \vee \texttt{Bool}){*})$. That is, when we apply `map(even)` to a list of integers it returns a list of booleans; when we apply it to a list that does not contain integers, then it returns a list of the same type; when the list contains some integers (*eg*, a list of reals), then it replaces integer elements by boolean ones. This is not an instance of the output type of `map`.

Acknowledgments. We were given a decisive help by Nino Salibra, who showed us how to prove that all powersets of infinite sets are convex models of the propositional logic. Pietro Abate implemented an early version of our prototype subtype checker. Bow-Yaw Wang drew our attention to the connection with convex theories. Claude Benzaken pointed out several useful results in combinatorics to us. Nils Gesbert spotted a subtle mistake in our subtyping algorithm. Many persons discussed the contents of this work with us and gave precious suggestions. Among them we are particularly grateful to Véronique Benzaken, Mariangiola Dezani-Ciancaglini, Kim Nguyen, Daniele Varacca, and Philip Wadler.

References

[1] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a web programming framework. In *ICFP '09*. ACM Press, 2009.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03*. ACM Press, 2003.

[3] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. *Theor. Comput. Sci.*, 398(1-3):217–242, 2008.

[4] G. Castagna and K. Nguyen. Typed iterators for XML. In *ICFP '08*, pages 15–26. ACM Press, 2008.

[5] J. Clark and M. Murata. Relax-NG, 2001. `www.relaxng.org`.

[6] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[7] D. Draper et al. XQuery 1.0 and XPath 2.0 Formal Semantics, 2007. `http://www.w3.org/TR/query-semantics/`.

[8] T. Berners-Lee et al. *Uniform Resource Identifier*, January 2005. RFC 3986, STD 66.

[9] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, Université Paris 7, December 2004.

[10] A. Frisch. OCaml + XDuce. In *ICFP '06*. ACM Press, 2006.

[11] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of ACM*, 55(4):1–64, 2008.

[12] V. Gapeyev, M.Y. Levin, B.C. Pierce, and A. Schmitt. The Xtatic compiler and runtime system, 2005. `http://www.cis.upenn.edu/~bcpierce/xtatic`.

[13] J. Garrigue. Programming with polymorphic variants. In *Proc. of ML Workshop*, 1998.

[14] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*. ACM Press, 2007.

[15] N. Gesbert, P. Genevès, and N. Layaïda. Parametric Polymorphism and Semantic Subtyping: the Logical Connection. In *ICFP '11*, 2011.

[16] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(2(7)):326–336, 1952.

[17] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. *ACM TOPLAS*, 32(1):1–56, 2009.

[18] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM TOIT*, 3(2):117–148, 2003.

[19] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.

[20] J.C. Mitchell. A type inference approach to reduction properties and semantics of polymorphic expressions. In *Lisp and Functional Programming*, pages 308–319, 1986.

[21] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, pages 513–523. Elsevier, 1983.

[22] J.C. Reynolds. Polymorphism is not set-theoretic. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 145–156. Springer, 1984.

[23] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL '06*, pages 103–114, 2006.

[24] W3C. *SOAP Version 1.2*. `http://www.w3.org/TR/soap`.

[25] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[26] C. Wallace and C. Runciman. Haskell and XML: Generic combinators or type based translation? In *ICFP '99*, pages 148–159, 1999.

[27] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03*, pages 224–235. ACM Press, 2003.

A. Termination of the subtyping algorithm

In this appendix we prove the termination of our subtyping algorithm and, hence, the decidability of the subtyping relation. The proof is done in two parts. First we prove that the algorithm terminates on finite types. This implies that the algorithm stops both on finite types *and* on infinite types that are not empty (since by Lemma 3.30, then there exists a finite expansion of the type that is not empty). Second, we prove that the algorithm terminates on all *empty* infinite types since it memoizes finitely many different types modulo alpha conversion. The bulk of the proof is proving that the algorithm terminates on finite types. Technically, this part of the proof proceeds in three steps. First, we introduce the notion of *saturated* set of types. More precisely a set of types $\mathscr{E} \subseteq \mathcal{T}$ is saturated if it satisfies some closure properties whose definition relies on the syntactic substitutions ($\{{}^{\neg\beta}/_{\alpha}\}$, $\{{}^{(\alpha_1\times\alpha_2)\vee\alpha_3}/_{\alpha}\}$, $\{{}^{(\alpha_1\to\alpha_2)\vee\alpha_3}/_{\alpha}\}$, $\{{}^{((\alpha_1\to\alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\vee\alpha_3}/_{\alpha}\}$) used by the algorithm (see Lemmas 3.10, 3.13, and 3.14). Second, we prove that the set of all types for which our decision algorithm terminates is saturated. Finally, we show that if a set $\mathscr{E}$ is saturated, then it contains the set $\mathcal{T}^{\mathsf{f}}$ of all finite types, that is $\mathscr{E} \supseteq \mathcal{T}^{\mathsf{f}}$. Whence we deduce that the decision algorithm terminates for all finite types. From a technical viewpoint thus, saturated sets are akin to *type-closed* set introduced by John Mitchell to prove strong normalization for second order typed λ-calculus [20].

Before giving the definition of saturated set we introduce some notations and give more details about the algorithm. In this section we use (possibly indexed) σ to range over syntactic substitutions always of the form used by the algorithm, that is $\{{}^{\neg\beta}/_{\alpha}\}$, $\{{}^{(\alpha_1\times\alpha_2)\vee\alpha_3}/_{\alpha}\}$, $\{{}^{((\alpha_1\to\alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\vee\alpha_3}/_{\alpha}\}$ or $\{{}^{(\alpha_1\to\alpha_2)\vee\alpha_3}/_{\alpha}\}$, and $\circ$ to denote their composition. We often use $\overline{\sigma}$ to denote a finite composition of substitutions, that is $\sigma_1 \circ \sigma_2 \circ ... \circ \sigma_n$. We also speak of subtrees of a type t. We intend by that the types that occur in the syntax tree of t, including t itself.

The decision algorithm for which we prove termination is essentially a dull implementation of the one presented in Section 2.6: it tries to check the emptiness of some type by trying to build a simulation containing it. It does so by decomposing the original problem into subproblems and using them to saturate the set obtained up to that point. More precisely, in order to check the emptiness of a type, the algorithm first normalizes it; it simplifies mixed intersections; it eliminates toplevel negative occurrences of variables by applying the substitution $\{{}^{\neg\beta}/_{\alpha}\}$; it eliminates toplevel positive occurrences of variables by applying $\{{}^{(\alpha_1\times\alpha_2)\vee\alpha_3}/_{\alpha}\}$, $\{{}^{(\alpha_1\to\alpha_2)\vee\alpha_3}/_{\alpha}\}$ or $\{{}^{((\alpha_1\to\alpha_2)\setminus(\mathbb{1}\to\mathbb{0}))\vee\alpha_3}/_{\alpha}\}$ substitutions; it eliminates the toplevel constructors by applying the equations of Definition 3.19 as left-to-right rewriting rules; the last point yields a set of different subproblems: the algorithm checks whether all the subproblems are solved (*ie*, it reached a simulation), otherwise it recurses on all of them. We introduce however a slight modification. While the algorithm described in Section 2.6 is defined for infinite types (and actually it works even if the types are non-regular), here we will use a finite representation of our types and use the μ-notation we introduced for the examples of Section 2.7. Since our (infinite) types are regular, then using μ-types (with the usual conditions of contractivity[13]) is equivalent, as proved by Courcelle [6]. This implies a small modification with respect to the algorithm of Section 2.6, since after the step of normalization the algorithm will unfold all the recursive types occurring at top-level. The use of the μ-notation will allow us to reason inductively on our types, although we will not be allowed to use induction on types in our main theorem, because of the unfolding performed after the normalization. The implementation we consider is "dull" insofar as it will explore all the subproblems to their end, even in the case when an answer could already be given earlier. So for example to check the emptiness of a disjunctive normal form, the algorithm will check the emptiness of *all* types in the union, even though it could stop as soon as it had found a non-empty type; similarly, to check the emptiness of $t_1 \times t_2$ the algorithm will check the emptiness of t_1 *and* of t_2, even though a positive results for, say, t_1 would make the check for t_2 useless. We do so because we want to prove that the algorithm is strongly normalizing, that is, termination does not depend on the reduction strategy.

Finally to count the steps performed by the algorithm we count the number of times the algorithm applies the rewriting rules that decompose toplevel constructors (*ie*, those of Definition 3.19). We must show that the algorithm performs finitely many steps for all finite types.

Definition A.1 (Saturated set)**.** *Let $\mathscr{E} \subseteq \mathcal{T}$ and let σ range over syntactic substitutions of the form $\{{}^{\neg\beta}/_{\alpha}\}$, $\{{}^{(\alpha_1 \times \alpha_2) \vee \alpha_3}/_{\alpha}\}$, $\{{}^{(\alpha_1 \to \alpha_2) \vee \alpha_3}/_{\alpha}\}$ or $\{{}^{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3}/_{\alpha}\}$. We say that $\mathscr{E}$ is* saturated *if $\forall n \geq 0$, $\forall \sigma_1 \ldots \sigma_n = \overline{\sigma}$ and $\forall t, t_1, t_2$:*

1. *$b \in \mathscr{E}$ (for all basic types b),*
2. *$\alpha\overline{\sigma} \in \mathscr{E}$ (for all type variables α),*
3. *if $\forall t_1'$ subtree of t_1, $\forall t_2'$ subtree of t_2, $t_1'\overline{\sigma} \in \mathscr{E}$ and $t_2'\overline{\sigma} \in \mathscr{E}$, then $t_1 \times t_2 \in \mathscr{E}$, $t_1 \to t_2 \in \mathscr{E}$, $t_1 \vee t_2 \in \mathscr{E}$ and $t_1 \wedge t_2 \in \mathscr{E}$.*
4. *if $\forall t'$ subtree of $t\{{}^{\mu x.t}/_{x}\}$, $t'\overline{\sigma} \in \mathscr{E}$, then $\mu x.t \in \mathscr{E}$,*
5. *if $\forall t'$ subtree of t, $t'\overline{\sigma} \in \mathscr{E}$, then $\neg t \in \mathscr{E}$,*

Next we prove that the set of the types for which the algorithm terminates is saturated. In order to do that we first need the following lemma.

Lemma A.2. *Let σ_i range over syntactic substitutions of the form $\{{}^{\neg\beta}/_{\alpha}\}$, $\{{}^{(\alpha_1 \times \alpha_2) \vee \alpha_3}/_{\alpha}\}$, $\{{}^{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3}/_{\alpha}\}$ or $\{{}^{(\alpha_1 \to \alpha_2) \vee \alpha_3}/_{\alpha}\}$. For all type variables α and substitutions $\sigma_1, ..., \sigma_n$ ($n \geq 0$) the algorithm terminates on all types of the form $\alpha\sigma_1...\sigma_n$.*

Proof. It is easy to see that all the types of the form $\alpha\sigma_1...\sigma_n$ are finite trees, and that they coincide with all the terms *inductively* generated by the following productions:

$$T ::= S \mid \neg S$$
$$S ::= \alpha \mid (T{\times}T) \vee T \mid (T{\to}T) \vee T \mid ((T{\to}T)\setminus(\mathbb{1}{\to}\mathbb{0})) \vee T$$

In words a type $\alpha\sigma_1...\sigma_n$ is always a possibly negated finite union of possibly negated variables, or possibly negated products or arrows (possibly minuse $\mathbb{1}{\to}\mathbb{0}$) of types of the same form. We are going to prove by induction on these terms that the algorithm terminates. Actually we prove a more general result namely that for all terms T inductively defined by the following productions (in bold the production we added)

$$T ::= S \mid \neg S$$
$$S ::= \alpha \mid (T{\times}T) \vee T \mid (T{\to}T) \vee T \mid ((T{\to}T)\setminus(\mathbb{1}{\to}\mathbb{0})) \vee T \mid \boldsymbol{T \wedge T}$$

[13] In our case we also have the condition that ensures finiteness of Boolean combinations, that is, that a recursion variable must be separated from its binder by at least one type constructor.

and for all $n > 0$, and $\sigma_1, ..., \sigma_n$, the algorithm terminates on $T\sigma_1...\sigma_n$.

Since the language of the latter productions includes that of the former ones, then the result is obtained as the special case of $n = 0$.

We prove this latter result for $T\sigma_1...\sigma_n$ by induction on the lexicographically ordered pairs (w, n), where n is the number of substitutions and w is the weight of syntax tree of T (recall that T is *inductively* defined) defined as follows: variables and basic types (the latter are not used in this lemma) have weight one, products and arrows sum the weights of their types, unions and intersections take the max of their weights, while the weight of the negation of a type is the weight of the type. Formally (recall that the types considered in this lemma are finite):

$$\begin{aligned}
\text{weight}(b) = \text{weight}(\alpha) &= 1 \\
\text{weight}(t_1{\times}t_2) = \text{weight}(t_1{\to}t_2) &= \text{weight}(t_1) + \text{weight}(t_2) \\
\text{weight}(t_1{\vee}t_2) = \text{weight}(t_1{\wedge}t_2) &= \max\{\text{weight}(t_1), \text{weight}(t_2)\} \\
\text{weight}(\neg t) &= \text{weight}(t)
\end{aligned}$$

This weight has the property that the application of normalization and of usual distributive laws of Boolean connectives, does not change the weight of a type.

(Case $n = 0$**)** We first consider the case $T = S$, that is with no outer negation. The first case for S is easy since if S is α, then the algorithm stops immediately returning false. If it is a union or a intersection then first the algorithm normalizes yielding a term of the following form (note that normalization does not change the weight):

$$\begin{aligned}
&\bigwedge_{i_1 \in P_1} (T^1_{i_1} \times T^2_{i_1}) \wedge \bigwedge_{j_1 \in N_1} \neg(T^1_{j_1} \times T^2_{j_1}) \\
\wedge &\bigwedge_{i_2 \in P_2} (T^1_{i_2} \to T^2_{i_2}) \wedge \bigwedge_{j_2 \in N_2} \neg(T^1_{j_2} \to T^2_{j_2}) \\
\wedge &\bigwedge_{i_3 \in \mathcal{V}_0} \alpha_{i_3} \wedge \bigwedge_{i_3 \in \mathcal{V}_1} \neg\alpha_{i_3}
\end{aligned}$$

If both P_1 and P_2 are not empty, then the algorithm immediately stops returning false. If just one of the two is non-empty, then the algorithm ignores the negated atoms of the other constructors and applies the substitutions $\overline{\sigma}$ to eliminate all top-level variables (both positive and negative). Next it applies the decomposition rules, that is, for every set $N' \subseteq N_1$, the algorithm checks the emptiness of

$$\bigwedge_{i_1 \in P_1} T^1_{i_1}\overline{\sigma} \wedge \bigwedge_{j_1 \in N'} \neg T^1_{j_1}\overline{\sigma} \wedge \bigwedge_{i_3 \in \mathcal{V}_0} \alpha^1_{i_3}$$

and

$$\bigwedge_{i_1 \in P_1} T^2_{i_1}\overline{\sigma} \wedge \bigwedge_{j_1 \in N_1 \setminus N'} \neg T^2_{j_1}\overline{\sigma} \wedge \bigwedge_{i_3 \in \mathcal{V}_0} \alpha^2_{i_3}$$

and similarly for arrow types.
The result follows by induction hypothesis since, even though n has increased, the weight w strictly decreased.

For the case $T = \neg S$, the easy cases are the one for negated variables (for which the algorithm stops immediately). For the negated intersections $\neg(T_1 \wedge T_2)$ since the algorithm has to check emptiness of $\neg T_1 \vee \neg T_2$ and since the weight does not change, this was already solved for unions earlier in this case. Similarly, the cases for unions are transformed into intersections and their case is deal exactly as we did for intersections earlier in this case.

(Case $n > 0$**)** This case is mostly straightforward and is obtained by separating the first substitution from the others $T\boldsymbol{\sigma_1}\sigma_2...\sigma_n$.

First notice that when all the substitutions are of the form $\{\neg\beta/_\alpha\}$, then the lemma follows straightforwardly by induction hypothesis: none of these substitutions changes the weight of the term and thus this becomes a case in which $n = 0$ (furthermore, the lemma trivially holds since we check the emptiness of a variable or of its negation). Thus the interesting case is when there is at least one substitution of the form $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/_\alpha\}$, $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3/_\alpha\}$ or $\{(\alpha_1 \to \alpha_2) \vee \alpha_3/_\alpha\}$. Without loss of generality we can consider that the first substitution σ_1 has one of these two forms, since otherwise we can move the first occurrence of such a substitution to the first position by applying its the preceding substitutions (which are just renamings and negations) to its image. So, for instance, if $\overline{\sigma} = \sigma_1 ... \sigma_n$ are all of the form $\{\neg\beta/_\alpha\}$ and σ_{n+1} is $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/_\alpha\}$, then we can remove $\overline{\sigma}$ and add in the first position $\{(\alpha_1\overline{\sigma} \times \alpha_2\overline{\sigma}) \vee \alpha_3\overline{\sigma}/_\alpha\}$.

Suppose that the first substitution σ_1 is $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/_\alpha\}$. If $T = \alpha$, then $T\sigma_1\sigma_2...\sigma_n = (\alpha_1\sigma_2...\sigma_n \times \alpha_2\sigma_2...\sigma_n) \vee \alpha_3\sigma_2...\sigma_n$. We can redo the proof of the corresponding case with $n = 0$ and apply the induction hypothesis since $\alpha_i\sigma_2...\sigma_n$ has the same weight as $\alpha\sigma_1...\sigma_n$ (since the weight(α) = weight$(\alpha_i) = 1$) but smaller n. If $T = \beta \neq \alpha$, then $T\sigma_1\sigma_2...\sigma_n = T\sigma_2...\sigma_n$ and so the result follows by induction. All the other cases are similar. The only subtle one is when $T = (T_1 \times T_2) \vee \alpha$ and α is not in the domain of any σ_i's, that is $T\overline{\sigma} = (T_1\overline{\sigma} \times T_2\overline{\sigma}) \vee \alpha$ (and similarly for arrows). In that case the algorithm eliminates the toplevel variable by performing the substitution $\sigma' = \{(\alpha_1 \times \alpha_2) \vee \alpha_3/_\alpha\}$ and by decomposing the intersection of the so obtained two top-level products into two subproblems of checking the emptiness of $T_i\overline{\sigma}\sigma' \vee \alpha_i$ for $i = 1, 2$. Since the α_i's are fresh, then both problems are equivalent to checking emptiness of $(T_i \vee \alpha_i)\overline{\sigma}\sigma'$, and since the weight of $(T_i \vee \alpha_i)$ is strictly smaller than that of T, then the result follows by induction hypothesis.

□

Although the result of the previous lemma is unsurprising, it is quite interesting since it gives us the first key to understand why the algorithm is strongly normalizing. It suggests that if during the execution we follow the evolution of an occurrence of a variable, then the test of this occurrence will never generate infinite types, but just finite ones. So substitutions by themselves and the new types they introduce cannot be the source of an infinite loop. So the only way to generate loops is by combining some variables with some subtree of the original tree. But the regularity of trees makes the number of these combinations finite, which is a consequence of the following theorem.

Theorem A.3. *Let $\mathcal{SN} = \{t \mid$ the algorithm terminates for $t\}$. $\mathcal{SN}$ is saturated.*

Proof. We associate to each type t in $\mathcal{SN}$ a rank n where n is the number of steps that the algorithm performs when fed with the type t (recall that a step is the application of a decomposition of a toplevel type constructor) and prove the theorem by induction on n. (Recall that we cannot use the induction on the structure of types because μ-types are unfolded after that they are normalized). More precisely, we have to prove that the five conditions of Definition A.1 are satisfied by $\mathcal{SN}$ and when considering types of a given rank we will assume by induction hypothesis that the five conditions are satisfied by all types of strictly smaller rank:

Case 1: it is trivial to show that for every basic type b we have that $b \in \mathcal{SN}$: the algorithm immediately stops returning false, as b is never empty.

Case 2: This case is proved by Lemma A.2.

Case 3: We suppose that $\forall t'_1$ subtree of t_1, $\forall t'_2$ subtree t_2, $t'_1\overline{\sigma} \in \mathcal{SN}$ and $t'_2\overline{\sigma} \in \mathcal{SN}$. We have to prove that $t_1 \times t_2$, $t_1 \to t_2$, $t_1 \vee t_2$, and $t_1 \wedge t_2$ are all in $\mathcal{SN}$. Let us consider the four subcases separately.

Subcase $(t_1 \times t_2)$: In this case the algorithm checks whether t_1 and t_2 are empty. Since we assumed that both are in $\mathcal{SN}$ (recall that we suppose that the set of subtrees of a type include the type itself), then the algorithm terminates also for $(t_1 \times t_2)$.

Subcase $(t_1 \to t_2)$: In this case the algorithm ends immediately since no arrow type is empty. Therefore $(t_1 \to t_2) \in \mathcal{SN}$.

Subcase $t_1 \vee t_2$: This case is similar to the one for $(t_1 \times t_2)$ since to check that a union is empty the algorithm checks whether each type is empty.

Subcase $t_1 \wedge t_2$: This is the difficult case. First of all notice that a type is in $\mathcal{SN}$ if and only if its disjunctive normal form is in $\mathcal{SN}$, too[14]. So without loss of generality we can consider just types in normal form. Thus, let

$$t_1 = \bigvee_{i_1 \in I_1} \bigwedge_{j_1 \in J_{i_1}} \ell_{i_1 j_1}$$

and

$$t_2 = \bigvee_{i_2 \in I_2} \bigwedge_{j_2 \in J_{i_2}} \ell_{i_2 j_2},$$

then the normal form of the intersection is

$$t_1 \wedge t_2 = \bigvee_{i_1 \in I_1, i_2 \in I_2} \bigwedge_{j_1 \in J_{i_1}} \ell_{i_1 j_1} \wedge \bigwedge_{j_2 \in J_{i_2}} \ell_{i_2 j_2}.$$

Since we have a union, then the algorithm checks the emptiness of each intersection $\bigwedge_{j_1 \in J_{i_1}} \ell_{i_1 j_1} \wedge \bigwedge_{j_2 \in J_{i_2}} \ell_{i_2 j_2}$, let us denote it by t_{ij}. We write t_{ij} in clearer form, by separating the different constructors and their negations

$$\begin{aligned} t_{ij} = & \bigwedge_{i_1 \in P_1} b_{i_1} \wedge \bigwedge_{j_1 \in N_1} \neg b_{j_1} \\ & \wedge \bigwedge_{i_2 \in P_2} (t^1_{i_2} \times t^2_{i_2}) \wedge \bigwedge_{j_2 \in N_2} \neg(t^1_{j_2} \times t^2_{j_2}) \\ & \wedge \bigwedge_{i_3 \in P_3} (t^1_{i_3} \to t^2_{i_3}) \wedge \bigwedge_{j_3 \in N_3} \neg(t^1_{j_3} \to t^2_{j_3}) \\ & \wedge \bigwedge_{i_4 \in \mathcal{V}_0} \alpha_{i_4} \end{aligned}$$

t_{ij} may contain some negative type variables. If it is the case then the various t^h_k are obtained from the literals of t_{ij} by applying one or several substitutions of the form $\{\neg\beta/_\alpha\}$, to eliminate toplevel negative occurrences of type variables. This is why we just considered toplevel positive occurrences of variables. We now proceed by an analysis of all possible cases:

- **at least two of P_1, P_2 and P_3 are not empty:** The algorithm stops immediately and returns true: t_{ij} will be empty since it contains different positive atoms. So $t_{ij} \in \mathcal{SN}$.
- **$|P_1| > 0$ and $|P_2| = |P_3| = 0$:** since the negative product types contain all the basic and arrow types, then we can erase them and rewrite t_{ij} as $\bigwedge_{i_1 \in P_1} b_{i_1} \wedge \bigwedge_{j_1 \in N_1} \neg b_{j_1} \wedge \bigwedge_{i_4 \in \mathcal{V}_0} \alpha_{i_4}$. According to Lemma 3.11, we can decide whether t_{ij} is empty or not, that is, $t_{ij} \in \mathcal{SN}$.

[14] Actually all its disjunctive normal forms, since the disjunctive normal form of a type is not unique

- $|P_2| > 0$ **and** $|P_1| = |P_3| = 0$**:** we can rewrite t_{ij} as

$$\bigwedge_{i_2\in P_2}(t^1_{i_2}\times t^2_{i_2})\wedge\bigwedge_{j_2\in N_2}\neg(t^1_{j_2}\times t^2_{j_2})\wedge\bigwedge_{i_4\in\mathcal{V}_0}\alpha_{i_4}.$$

If $|\mathcal{V}_0| = 0$, following Lemma 3.15, for any set $N'\subseteq N_2$, the algorithm will check

$$\bigwedge_{i_2\in P_2} t^1_{i_2}\wedge\bigwedge_{j_2\in N'}\neg t^1_{j_2}$$

and

$$\bigwedge_{i_2\in P_2} t^2_{i_2}\wedge\bigwedge_{j_2\in N_2\setminus N'}\neg t^2_{j_2}$$

Since every $(t^1_i\times t^2_i)$ is from t_1 or t_2 ($i\in P_2\cup N_2$), then all t^1_i's and t^2_i's are subtrees of t_1 or t_2. Let t' be any subtree of t^1_i. Then t' is also a subtree of t_1 or t_2. Following the premise, we have $t'\overline{\sigma}\in\mathcal{SN}$ for all $\overline{\sigma}$. Furthermore we also assumed that both t^1_i and t^2_i are in $\mathcal{SN}$. To check the emptiness of $(t^1_i\times t^2_i)$, the algorithm checks the emptiness of both t^1_i and t^2_i. Then not only $(t^1_i\times t^2_i)$ is strongly normalizing, but also both t^1_i and t^2_i normalize in strictly less steps than $(t^1_i\times t^2_i)$. So we can use the induction hypothesis for t^1_i and t^2_i and assume that the conditions of saturation are satisfied for them. In particular since we have that for every substitution $\overline{\sigma}$ and subtree t' of theirs, $t'\overline{\sigma}\in\mathcal{SN}$, then we can apply the case 5 of Definition A.1 and deduce that also $\neg t^1_i\in\mathcal{SN}$. Similarly, by applying case 3 of Definition A.1 we obtain by induction that $\bigwedge_{i_2\in P_2} t^1_{i_2}\wedge\bigwedge_{j_2\in N'}\neg t^1_{j_2}\in\mathcal{SN}$ and $\bigwedge_{i_2\in P_2} t^2_{i_2}\wedge\bigwedge_{j_2\in N_2\setminus N'}\neg t^2_{j_2}\in\mathcal{SN}$. So t_{ij} is in $\mathcal{SN}$ as well.

If $|\mathcal{V}_0| > 0$, the algorithm performs the substitutions $\sigma_i = \{(\alpha^1_i\times\alpha^2_i)\vee\alpha^3_i/_\alpha\}$ for $i\in\mathcal{V}_0$ (where α^1_i, α^2_i, and α^3_i are fresh type variables) yielding

$$t_{ij}=\bigwedge_{i_2\in P_2}(t^1_{i_2}\overline{\sigma'}\times t^2_{i_2}\overline{\sigma'})\wedge\bigwedge_{j_2\in N_2}\neg(t^1_{j_2}\overline{\sigma'}\times t^2_{j_2}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}_0}(\alpha^1_{i_4}\times\alpha^2_{i_4})$$

where $\overline{\sigma'}=\sigma_1\ldots\sigma_m$ ($i\in\mathcal{V}_0$). Following Lemma 3.15, for any set $N'\subseteq N_2$, the algorithm will check

$$(\bigwedge_{i_2\in P_2} t^1_{i_2}\overline{\sigma'}\wedge\bigwedge_{j_2\in N'}\neg t^1_{j_2}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}_0}\alpha^1_{i_4}$$

and

$$(\bigwedge_{i_2\in P_2} t^2_{i_2}\overline{\sigma'}\wedge\bigwedge_{j_2\in N_2\setminus N'}\neg t^2_{j_2}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}_0}\alpha^2_{i_4}$$

Since all the t^1_i's and t^2_i's are subtrees of t_1 or t_2, then for every t' subtree of the $t^1_i\overline{\sigma'}$'s or $t^2_i\overline{\sigma'}$'s we have $t'\overline{\sigma}\in\mathcal{SN}$: the t' will always come from a part of t^1_i, to which we have applied a substitution of the expected form. Furthermore, we also know that all variables are in $\mathcal{SN}$. So we can proceed as in the case in which $|\mathcal{V}_0| = 0$, and deduce the result by applying the induction hypothesis.

- $|P_3| > 0$ **and** $|P_1| = |P_2| = 0$**:** similarly to the previous case, we can rewrite t_{ij} as

$$\bigwedge_{i_3\in P_3}(t^1_{i_3}\to t^2_{i_3})\wedge\bigwedge_{j_3\in N_3}\neg(t^1_{j_3}\to t^2_{j_3})\wedge\bigwedge_{i_4\in\mathcal{V}_0}\alpha_{i_4}.$$

If $|\mathcal{V}_0| = 0$, following Lemma 3.16, for any $j_3\in N_3$ and any set $P'\subseteq P_3$, the algorithm will check

$$\bigwedge_{i_3\in P'}\neg t^1_{i_3}\wedge t^1_{j_3}$$

and

$$\bigwedge_{i_3\in P_3\setminus P'} t^2_{i_3}\wedge\neg t^2_{j_3}$$

Similarly to **Case** $|P_2| > 0$ **and** $|P_1| = |P_3| = 0$, by induction, we can prove $\bigwedge_{i_3\in P'}\neg t^1_{i_3}\wedge t^1_{j_3}\in\mathcal{SN}$ and $\bigwedge_{i_3\in P_3\setminus P'} t^2_{i_3}\wedge\neg t^2_{j_3}\in\mathcal{SN}$. So t_{ij} is in $\mathcal{SN}$, as well.

If $|\mathcal{V}_0| > 0$, the algorithm performs the substitutions $\sigma_i = \{(\alpha^1_i\to\alpha^2_i)\vee\alpha^3_i/_\alpha\}$ or $\{((\alpha^1_i\to\alpha^2_i)\setminus(\mathbb{1}\to\mathbb{0}))\vee\alpha^3_i/_\alpha\}$ for $i\in\mathcal{V}_0$ (where α^1_i, α^2_i, and α^3_i are fresh type variables) yielding

$$t_{ij}=\bigwedge_{i_3\in P_3}(t^1_{i_3}\overline{\sigma'}\to t^2_{i_3}\overline{\sigma'})\wedge\bigwedge_{j_3\in N_3}\neg(t^1_{j_3}\overline{\sigma'}\to t^2_{j_3}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}_0}(\alpha^1_{i_4}\to\alpha^2_{i_4})(\wedge\neg(\mathbb{1}\to\mathbb{0}))$$

where $\overline{\sigma'}=\sigma_1\ldots\sigma_m$ ($i\in\mathcal{V}_0$). Since the top-level variables are finite, then the possible substitutions are also finite ($2^{|\mathcal{V}_0|}$). According to Lemma 3.16, for any $j_3\in N_3$, any set $P'\subseteq P_3$ and any set $\mathcal{V}'\subseteq\mathcal{V}_0$, the algorithm will check

$$(\bigwedge_{i_3\in P'}\neg t^1_{i_3}\overline{\sigma'}\wedge t^1_{j_3}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}'}\neg\alpha^1_{i_4}$$

and

$$(\bigwedge_{i_3\in P_3\setminus P'} t^2_{i_3}\overline{\sigma'}\wedge\neg t^2_{j_3}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}_0\setminus\mathcal{V}'}\alpha^2_{i_4}$$

Note that $(\bigwedge_{i_3\in P'}\neg t^1_{i_3}\overline{\sigma'}\wedge t^1_{j_3}\overline{\sigma'})\wedge\bigwedge_{i_4\in\mathcal{V}'}\neg\alpha^1_{i_4}$ contains some top-level negative type variables. By applying one or several substitutions of the form $\{\neg\beta/_\alpha\}$, we eliminate these toplevel negative occurrences of type variables. That means the algorithm will check $(\bigwedge_{i_3\in P'}\neg t^1_{i_3}\overline{\sigma'\sigma''}\wedge t^1_{j_3}\overline{\sigma'\sigma''})\wedge\bigwedge_{i_4\in\mathcal{V}'}\alpha^1_{i_4}$ instead. The rest of the proof proceeds as we did in **Case** $|P_2| > 0$ **and** $|P_1| = |P_3| = 0$.

- $|P_1| = |P_2| = |P_3| = 0$**:** If at least one of N_1, N_2 of N_3 is empty, then the algorithm stops immediately (returning false, since the negation of one or two constructors contains all the types of the other missing constructors). Otherwise, if $|N_i| > 0$ for $i = 1, 2, 3$, then the algorithm checks separately whether $\mathbb{1}_{\mathscr{C}}\wedge\bigwedge_{j_1\in N_1}\neg b_{j_1}$ (where $\mathbb{1}_{\mathscr{C}}$ denotes a basic type that contains all the constants $\mathscr{C}$, that is $\mathbb{1}\wedge\neg(\mathbb{1}\times\mathbb{1})\wedge\neg(\mathbb{0}\to\mathbb{1})$) is empty, that is the case of $|P_1| > 0$ **and** $|P_2| = |P_3| = 0$; whether $(\mathbb{1}\times\mathbb{1})\wedge\bigwedge_{j_2\in N_2}\neg(t^1_{j_2}\times t^2_{j_2})$ is empty, that is the case of $|P_2| > 0$ **and** $|P_1| = |P_3| = 0$; and whether $(\mathbb{0}\to\mathbb{1})\wedge\bigwedge_{j_3\in N_3}\neg(t^1_{j_3}\to t^2_{j_3})$ is empty, that is the case of $|P_3| > 0$ **and** $|P_1| = |P_2| = 0$. It returns true only if all them succeed: indeed the only case in which an intersection of negated types is empty is when the basic type component is $\mathbb{1}_{\mathscr{C}}$, the product component is $\mathbb{1}\times\mathbb{1}$, and the arrow component is $\mathbb{0}\to\mathbb{1}$.

Case 4: If $t=\mu x.t'$, then we have two subcases. Either x occurs free in t'. In which case $\mu x.t'$ is a subtree of $t'\{\mu x.t'/_x\}$. Therefore the result follows directly from the assumption that every subtree of $t'\{\mu x.t'/_x\}$ is in $\mathcal{SN}$. Or x does not occur free in t', then $t=t'$ and the result will be obtained by applying the case of the proof corresponding to the form of t': just notice that by contractivity there may not be an infinite sequence of μ-abstractions: eventually the recursion must traverse a type constructor.

Case 5: Let $t=\bigvee_{i\in I}\bigwedge_{j\in J_i}\ell_{ij}$ and suppose that for all subtrees t' of t and finite composition of substitutions $\overline{\sigma}$, also $t'\overline{\sigma}$ is in $\mathcal{SN}$.

To prove this case we must prove that $\neg t \in \mathcal{SN}$, as well. The normal form of $\neg t$ is:

$$\bigvee_{j_1 \in J_{i_1}, \ldots, j_n \in J_{i_n}} \neg \ell_{i_1 j_{i_1}} \wedge \ldots \wedge \neg \ell_{i_n j_{i_n}}$$

where $I = \{i_1, \ldots, i_n\}$. Therefore the algorithm checks whether each $\neg \ell_{i_1 j_{i_1}} \wedge \ldots \wedge \neg \ell_{i_n j_{i_n}}$ is empty. To prove that all these checks terminate it suffices to proceed as in **Subcase** $t_1 \wedge t_2$ of **Case** 3: we consider all the possible forms of the literals in the intersection and apply the corresponding decomposition rules. Once more the result follows by induction. □

Finally, if a set is saturated, then it contains all finite types.

Theorem A.4. *If $\mathscr{E} \subseteq \mathcal{T}$ is saturated, then $\mathcal{T}^{\mathsf{f}} \subseteq \mathscr{E}$*

Proof. The intuition is that since a saturated set contains all basic variables, all type variables and it is closed for all constructors, then it contains all types. Formally, we prove a (apparently) stronger result, namely that if $\mathscr{E}$ is saturated, then $\forall n \geq 0$, $\forall \sigma_1 \ldots \sigma_n = \overline{\sigma}$ (where, as before, σ denotes substitution of the form $\{\neg\beta/_{\alpha}\}$, $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/_{\alpha}\}$, $\{(\alpha_1 \to \alpha_2) \vee \alpha_3/_{\alpha}\}$) or $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3/_{\alpha}\}$), and $\forall t \in \mathcal{T}^{\mathsf{f}}$, the type $t\overline{\sigma}$ is in $\mathscr{E}$.

Without loss of generality we can consider $\mathcal{T}^{\mathsf{f}}$ as the set of types that do not contain any subtree of the form $\mu x.t$ (if a finite trees contains such a subtree, then the μ-constructor can be easily eliminated). We proceed by induction on the structure of t. The base cases follow immediately from the first two cases of the definition of saturation, which state that $\mathscr{E}$ contains b (and therefore $b\overline{\sigma}$) and $\alpha\overline{\sigma}$, for all basic types b and type variables α.

If $t = (t_1 \times t_2)$ then by induction hypothesis $\forall t_1'$ subtree of t_1 and $\forall t_2'$ subtree t_2, $t_1'\overline{\sigma} \in \mathscr{E}$ and $t_2'\overline{\sigma} \in \mathscr{E}$. Therefore we can apply the third case of the definition of saturation and deduce that $(t_1 \times t_2) \in \mathscr{E}$. Now take a new substitution $\overline{\sigma'}$ and consider $t\overline{\sigma'}$, that is $(t_1\overline{\sigma'} \times t_2\overline{\sigma'})$: all it remains to prove is that $t\overline{\sigma'} \in \mathscr{E}$. It is easy to see that every subtree of $t_i\overline{\sigma'}$ $(i = 1, 2)$ is of the form $t_i'\overline{\sigma'}$ with t_i' subtree of t_i. But since $\forall t_i'$ subtree of t_i and all $\overline{\sigma}$ one has $t_i'\overline{\sigma} \in \mathscr{E}$, then in particular $t_i'\overline{\sigma'}\,\overline{\sigma''} \in \mathscr{E}$ for all $\overline{\sigma''}$. So we can once more apply the case 3 of Definition A.1, deduce that $(t_1\overline{\sigma'} \times t_2\overline{\sigma'}) \in \mathscr{E}$, which corresponds to state that $t\overline{\sigma'} \in \mathscr{E}$, that is, the result. The cases in which t is $t_1 \to t_2$, $t_1 \vee t_2$, $t_1 \wedge t_2$, of $\neg t$ are similar.

Notice that the proof of this theorem cannot be extended to the case for t of the form $\mu x.t$ since we cannot apply the induction hypothesis on its unfolding. □

Corollary A.5. *The algorithm terminates on all finite types.*

We can combine the result of this corollary with the result of Lemma 3.30 to obtain that the algorithm terminates also on infinite non-empty types: if an infinite type is non-empty, then by Lemma 3.30 there exists a finite expansion of it which is not empty and on which, by Corollary A.5 the algorithm terminates (provided that the algorithm implements a breadth-first search).

Corollary A.6. *The algorithm terminates on all non-empty types.*

Finally, the algorithm terminates also on empty infinite types. Here the point is simple. During its execution the algorithm checks the emptiness of several types: the previous result assures that the check terminates on non-empty types, while for the types that are empty, the algorithm terminates because it has to check only finitely many different options. The latter is a consequence of the regularity of the types and the fact that all the types that are checked are (a finite combination of) subterms of the original type to which substitutions all of the same form are applied, substitutions that we (morally) keep symbolical. The algorithm works coinductively and memoizes the intermediate types it is considering. Before memoizing and recursing however the system checks whether the current type is already memoized or, by applying the Lemma A.8 below, *it is an instance of* a memoized type in which case it stops. The only case in which the algorithm might diverge is by generating a sequence of empty types and memoizing them, but this does not happen since eventually this chain produces *an instance* of a type previously occurring in the chain.

Lemma A.7. *Let t, t' be two types*

$$\forall \eta \,.\, \mathbb{E}(t)\eta = \varnothing \Rightarrow \forall \eta \,.\, \mathbb{E}(t \wedge t')\eta = \varnothing$$

Proof. Straightforward since $\mathbb{E}(t \wedge t')\eta = \mathbb{E}(t)\eta \cap \mathbb{E}(t')\eta$. □

Lemma A.8. *Let t be a type. Then*

$$\forall \eta \,.\, \mathbb{E}(t)\eta = \varnothing \Rightarrow \forall \eta \,.\, \mathbb{E}(t\overline{\sigma})\eta = \varnothing$$

where $\overline{\sigma} = \forall n \geq 0 \,.\, \sigma_1 \ldots \sigma_n$ and σ_i denotes substitution of the form $\{\neg\beta/_{\alpha}\}$, $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/_{\alpha}\}$, $\{(\alpha_1 \to \alpha_2) \vee \alpha_3/_{\alpha}\}$ or $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3/_{\alpha}\}$.

Proof. An application of Lemma 3.20. □

Theorem A.9. *The algorithm terminates on all types.*

Proof. Consider a generic type t. If t is a finite type, then according to Corollary A.5, the algorithm terminates. Assume that t is an infinite type (recursive type). If t is nonempty, then by Corollary A.6 the algorithm terminates, too. The only remaining case is when t is an *empty* infinite type. Let $\mathcal{N}(t)$ be a disjunctive normal form of t. According to set theory, $\mathcal{N}(t)$ is empty if and only if all its single normal forms are empty. Let us just consider one of its single normal forms.

To check the emptiness of t, the algorithm first checks whether t is memoized (*ie*, whether this type was already met during the check and is therefore supposed to be empty), or t is an instance of a type t' that is memoized (see Lemma A.8), or t from which we removed some atom intersections is an instance of a memoized type (this step is correct by Lemma A.7). If it is, then the algorithm terminates, otherwise, it memoizes t.[15] Next if $|\mathsf{tlv}(t)| > 0$, then the algorithm performs the appropriate substitution(s) (see Lemmas 3.10, 3.13, and 3.14). Then according to the decomposition rules (see Lemmas 3.15 and 3.16), the algorithm decomposes t into several types which are the candidate types to be checked on the next iteration. t is empty if and only if some of the candidate types (depending on the decomposition rules) are empty. If a candidate is not empty, then the algorithm stops on that candidate (Corollary A.6), otherwise the algorithm reiterates the memoization process. This iteration (performed on empty candidates) eventually stops on a memoized term. To see why let us consider the form of all types that can be met during the checking of the emptiness of t (that we suppose to be empty). These are single normal forms (*cf.* Definition 3.8), that is, intersections of atoms. Now in any of these intersections we can distinguish two parts: there is a part of the intersection that is formed just by type variables (these are either the variables of the original type or some fresh variables that were introduced to eliminate a toplevel variable), and a second part that intersects basic and/or product and/or arrow types. If the check of memoization fails, then the first part of the type formed by the intersection of variables is eliminated, and the appropriate substitution(s) is (are) applied to the second part. Then the atoms in this second part to which the substitution(s) is applied are decomposed

[15] If t happens to be nonempty, then in the real algorithm t will be removed from the memoized set, but this is just an optimization that reduces the number of required checks and does not affect the final result.

to form the next set of single normal forms. It is then clear that the second part of all the candidate single normal forms met by the algorithm are formed by chunks of the original type to which a succession of substitutions of the same form as those used in Lemmas 3.10, 3.13, and 3.14 is applied. So we can formally characterize all the single normal forms examined by the algorithm when checking the emptiness of a type t.

First, consider the original type t (for the sake of simplicity we will next give full details only for the case in which $t \leq \mathbb{1} \times \mathbb{1}$, but the proof is the same in general cases, as well). Next consider the set of all subtrees of t: since t is regular, then this set is finite. Finally consider the set $\mathcal{C}$ of all Boolean combinations of terms of the previous sets (actually, just single normal forms would suffice): modulo normalization (or modulo type semantics) there are only finitely many distinct combinations of a finite set of types, therefore $\mathcal{C}$ is finite as well. It is clear from what we said before that all the types that will be considered during the emptiness check for t will be of the form

$$\begin{array}{l}(t'\wedge\beta_1\wedge\ldots\wedge\beta_h)\{(\alpha_1^1\times\alpha_1^2)\vee\alpha_1^3/_{\alpha_1}\}\ldots\{(\alpha_n^1\times\alpha_n^2)\vee\alpha_n^3/_{\alpha_n}\}\\ \wedge\gamma_1\wedge\ldots\wedge\gamma_p\end{array} \quad (21)$$

where $t' \in \mathcal{C}$, $h, n, p \geq 0$, $\alpha_i \in \mathsf{var}(t') \cup \{\alpha_j^1, \alpha_j^2, \alpha_j^3 | 1 \leq j \leq i-1\}$, $\{\beta_i | 1 \leq i \leq h\} \cup \{\gamma_i | 1 \leq i \leq p\} \subseteq \{\alpha_i^1, \alpha_i^2, \alpha_i^3 | 1 \leq i \leq n\}$, $\{\beta_i | 1 \leq i \leq h\} \subseteq \{\alpha_i | 1 \leq i \leq n\}$, and $\{\gamma_i | 1 \leq i \leq p\} \cap \{\alpha_i | 1 \leq i \leq n\} = \varnothing$ (note that the γ_i's could be moved in the scope of the substitution, but we prefer to keep them separated for the time being).

Let us now follow one sequence of the check in which all the checked types are empty (since this is the only case in which the algorithm might diverge) and imagine by contradiction that this sequence is infinite. All the types in the sequence are of the form described in (21). Since $\mathcal{C}$ is finite, then there will be in the sequence a type t' occurring infinitely many times. Let s_1 and s_2 be two single normal forms in the sequence containing this particular t', namely:

$$\begin{array}{rcl} s_1 & = & (t' \wedge \beta_1 \wedge \ldots \wedge \beta_h)\{(\alpha_1^1\times\alpha_1^2)\vee\alpha_1^3/_{\alpha_1}\}\ldots \\ & & \{(\alpha_n^1\times\alpha_n^2)\vee\alpha_n^3/_{\alpha_n}\} \wedge \gamma_1 \wedge \ldots \wedge \gamma_p \\ s_2 & = & (t' \wedge \beta_1 \wedge \ldots \wedge \beta_k)\{(\alpha_1^1\times\alpha_1^2)\vee\alpha_1^3/_{\alpha_1}\}\ldots \\ & & \{(\alpha_m^1\times\alpha_m^2)\vee\alpha_m^3/_{\alpha_m}\} \wedge \gamma_1 \wedge \ldots \wedge \gamma_q \end{array}$$

Since we are checking the emptiness of these two types then the types can be considered modulo α-renaming of their variables. This justifies the fact that, without loss of generality, in the two terms above we can consider the first $\min\{n, m\}$ substitutions, the first $\min\{h, k\}$ β-variables and the first $\min\{p, q\}$ γ variables to be the same in both terms.

Let us consider again the infinite sequence of candidates that are formed by t' and consider the three cardinalities of the β variables, of the substitutions, and of the γ variables. Since $\mathbb{N}^3$ with a point-wise order is a well-quasi-order, we can apply Higman's Lemma [16] to this sequence and deduce that in the sequence there occur two types as the s_1 and s_2 above such that s_1 occurs before s_2 and $n \leq m$, $h \leq k$ and $p \leq q$.

Let us write σ_i^j for the substitution

$$\{(\alpha_i^1\times\alpha_i^2)\vee\alpha_i^3/_{\alpha_i}\}\ldots\{(\alpha_j^1\times\alpha_j^2)\vee\alpha_j^3/_{\alpha_j}\}$$

with $i \leq j$. We have that s_2 is equal to

$$(t'\wedge\beta_1\wedge\ldots\wedge\beta_h\wedge\ldots\wedge\beta_k)\sigma_1^m\wedge\gamma_1\wedge\ldots\wedge\gamma_p\wedge\ldots\wedge\gamma_q$$

we exit the rightmost β's, yielding:

$$(t'\wedge\beta_1\wedge\ldots\wedge\beta_h)\sigma_1^m\wedge\gamma_1\wedge\ldots\wedge\gamma_p\wedge\ldots\wedge\gamma_q\wedge(\beta_{h+1}\wedge\ldots\wedge\beta_k)\sigma_1^m$$

since the γ are independent from the α's we can move the leftmost ones inside a part of the substitutions obtaining

$$\begin{array}{l}((t'\wedge\beta_1\wedge\ldots\wedge\beta_h)\sigma_1^n\wedge\gamma_1\wedge\ldots\wedge\gamma_p)\sigma_{n+1}^m\\ \wedge\gamma_{p+1}\wedge\ldots\wedge\gamma_q\wedge(\beta_{h+1}\wedge\ldots\wedge\beta_k)\sigma_1^m\end{array}$$

which by definition of s_1 is equal to

$$s_1\sigma_{n+1}^m\wedge\gamma_{p+1}\wedge\ldots\wedge\gamma_q\wedge(\beta_{h+1}\wedge\ldots\wedge\beta_k)\sigma_1^m$$

In conclusion s_2 has the following form:

$$s_2 = s_1\sigma_{n+1}^m\wedge\gamma_{p+1}\wedge\ldots\wedge\gamma_q\wedge\beta_{h+1}\sigma_1^m\wedge\ldots\wedge\beta_k\sigma_1^m$$

therefore it is an intersection a part of which is an instance of the (memoized) type s_1. Therefore the algorithm (and thus the sequence) should have stopped on the check of s_2, which contradicts the hypothesis that the sequence is infinite.

□

In order to understand how the algorithm actually terminates on empty infinite types, consider for instance the following type:

$$\alpha \wedge (\alpha \times x) \wedge \neg(\alpha \times y)$$

where $x = (\alpha \wedge (\alpha \times x)) \vee \mathsf{nil}$ and $y = (\alpha \times y) \vee \mathsf{nil}$. First, the algorithm memoizes it. By an application of Lemma 3.13, the algorithm performs the substitution yielding

$$(\alpha_1 \times \alpha_2) \wedge (((\alpha_1 \times \alpha_2) \vee \alpha_3) \times x) \wedge \neg(((\alpha_1 \times \alpha_2) \vee \alpha_3) \times y).$$

Following Lemma 3.15, the algorithm checks the candidate types as follows:

$$\begin{cases} \begin{cases} \alpha_1 \wedge ((\alpha_1 \times \alpha_2) \vee \alpha_3) = \mathbb{0} & (1) \\ \text{or} \\ \alpha_2 \wedge x \wedge \neg y = \mathbb{0} & (2) \end{cases} \\ \text{and} \\ \begin{cases} \alpha_1 \wedge ((\alpha_1 \times \alpha_2) \vee \alpha_3) \wedge \neg((\alpha_1 \times \alpha_2) \vee \alpha_3) = \mathbb{0} & (3) \\ \text{or} \\ \alpha_2 \wedge x = \mathbb{0} & (4) \end{cases} \end{cases}$$

Type (1) is finite and nonempty, and type (3) is finite and empty. It is not necessary to check type (4) insofar as one of its expansions, $\alpha_1 \wedge \mathsf{nil}$, is not empty. So the algorithm terminates on type (4) as well. Considering type (2), it is neither memoized nor an instance of a memoized type, so it is memoized as well. Then the algorithm unfolds it and gets

$$\begin{array}{l}\alpha_2 \wedge ((\alpha_1\times\alpha_2) \vee \alpha_3) \wedge (((\alpha_1\times\alpha_2) \vee \alpha_3) \times x)\\ \wedge \neg(((\alpha_1\times\alpha_2) \vee \alpha_3) \times y)\end{array}$$

The algorithm matches the unfolded type with the memoized ones. It is an instance of

$$\alpha_2 \wedge \alpha \wedge (\alpha \times x) \wedge \neg(\alpha \times y)$$

where the substitution is $\{((\alpha_1\times\alpha_2) \vee \alpha_3)/_{\alpha}\}$. Although it is not memoized, it can be deduced to be empty from the memoized $\alpha \wedge (\alpha \times x) \wedge \neg(\alpha \times y)$ and Lemma A.7.