Semantic Subtyping for Objects and Classes

Ornela Dardha Daniele Gorla Daniele Varacca

Abstract

In this paper we propose an integration of structural subtyping with boolean connectives and semantic subtyping to define a Java-like programming language that exploits the benefits of both techniques. Semantic subtyping is an approach for defining subtyping relation based on set-theoretic models, rather than syntactic rules. On the one hand, this approach involves some non trivial mathematical machinery in the background. On the other hand, final users of the language need not know this machinery and the resulting subtyping relation is very powerful and intuitive. While semantic subtyping is naturally linked to the structural one, we show how our framework can also accommodate the nominal subtyping. Several examples show the expressivity and the practical advantages of our proposal.

1 Introduction

A *type system* for a programming language is a set of deduction rules that enable type derivations for the terms of the language. The *subtyping relation* on types is a key notion as type systems often depend on it. There are two main approaches for defining the subtyping relation: the *syntactic* approach and the *semantic* one. The syntactic approach is more common; it is defined by means of a formal system of deduction rules. One proceeds as follows: first define the language, then the set of syntactic types and finally the subtyping relation by deduction rules. In the semantic approach, instead, one starts from a model of the language and an interpretation of types as subsets of this model; the subtyping relation is then defined as inclusion of sets denoting types. This approach has received less attention than the syntactic one because it is more technical: it is not trivial to define the interpretation of types as subsets of a model, or to define a model at all. However, the semantic approach presents several advantages: it allows a natural definition of boolean operators and the meaning of types is more intuitive for the programmer, who does not need to know the theory behind the curtain (as usual in sophisticated programming environments).

The first use of the semantic approach goes back to two decades ago [1, 2]. More recently, Hosoya and Pierce [3, 4] have adopted this approach to define XDuce, an XML-oriented language which transforms XML documents into other XML documents, satisfying certain properties. Subtyping relation is established as inclusion of sets of values, the latter being fragments of XML documents. The type system contains booleans, products and recursive types. There are no function types and terms in the language. Benzaken, Castagna and Frisch [5, 6, 7] extend the XDuce with first-class functions and arrow types defining a higher-order language, named $\mathbb{C}$Duce, and adopting the semantic approach to subtyping. The starting point of their framework is a higher-order $\lambda-$calculus with pairs and projections. The set of types is extended with intersection, union and negation types interpreted in a set-theoretic way. More recently the type system of $\mathbb{C}$Duce was also extended to include polymorphism [8].

This approach can also be applied to π-calculus [9]. Castagna et al. [10] defined the $\mathbb{C}\pi$ language, a variant of the asynchronous π-calculus where channel types are augmented with boolean connectives.

Amadio and Cardelli [11] define subtyping for recursive types for the λ-calculus. Types are interpreted as collections of values and subtyping corresponds to subcollections. They introduce the notion of bottom and top types. Types are partially ordered by using the inclusion relation and every type t is in relation of subtyping with bottom and top, as the intuition conveys. In this work, Amadio and Cardelli extend this ordering to recursive types: if one assumes that the inclusion of recursive variables implies inclusion of the bodies of recursive types, one can deduce the inclusion of recursive types themselves. They show that subtyping in the presence of recursive types is decidable, but did not provide a complexity analysis. Later on Kozen et al. [12] show that the type inclusion problem is solvable in time $O(n^2)$. They reduce the problem to the emptiness problem for automata.

Aiken and Wimmers [13] present an algorithm for solving systems of type inclusion constraints, where the type language includes 0 and 1, being the least and universal types, respectively, intersection and union types, function types, pairs and recursive types. This algorithm stands at the basis of an inclusion-based type inference system for the λ-calculus with constants.

Ancona and Lagorio [14] study the subtyping relation for infinite types defined coinductive by using union and object type constructors. Types are interpreted as sets of values, by exploiting the notions of membership and contractive derivations. Subtyping is defined coinductively and is shown to be sound w.r.t. set inclusion of values. This technique eliminates the circularity and the need for a bootstrap model. This differs from our work in both the interpretation of types and the definition of subtyping. A recent alternative semantic approach to subtyping for record types is by Ancona and Corradi [15], who concentrate on the use of coinductively defined types to model cyclic record values.

Bonsangue et al. [16] develop a coalgebraic approach to coinductive types. The paper defines a set-theoretic interpretation of coinductive types with union types, and defines the semantic subtyping relation as inclusion of maximal traces, which is syntactic unfoldings of type definitions. Morevover, a technique is proposed to define subtyping as inclusion of merely finite traces. The technique proposed in this work is completely different from ours. In addition, our approach based on set-inclusion of values is applied to a specific framework, that of an object-oriented language. Other relevant related works on coinductive definitions of subtyiping are [17, 18] and the recent survey [19].

Finally, semantic subtyping is adopted in a flow-typing calculus [20]. Flow-typing allows a variable to have different types in different parts of a program and thus is more flexible than the standard static typing. Type systems for flow-typing incorporate intersection, union and negation types in order to typecheck terms, like for example, *if-then-else* statements. Consequently, semantic subtyping is naturally defined on top of these systems.

Contributions. In the present paper, we address the semantic subtyping approach (§ 3) by applying it to an object-oriented core language; the result will be an object-oriented system with boolean connectives and structural subtyping. To this aim, we use the syntax of *Featherweight Java* (FJ) [21], which is a functional fragment of Java (§ 2). We give the syntax of types (§ 2.1) and terms (§ 2.2), and the operational semantics

(§ 6) of our calculus. We define a type system (§ 4) that uses a subtyping relation, as we discuss in the following, and prove its safety by the usual subject reduction (Thm 2, § 7) and progress (Thm 3, § 7).

From a technical point of view, the development is not trivial. It follows [7], but with the difference that we do not have higher-order values. Therefore, we cannot directly reuse their results. Instead, we define and construct from scratch the semantic model, called *bootstrap model* (§ 3.3), which induces the subtyping relation used in the type system of our language. We prove some important theoretical results regarding this model (the details are given in the Appendix). After having built the bootstrap model and defined the subtyping relation it induces, we define a new and more natural interpretation of types, as sets of (pseudo-)values (§ 5). The reason is that we want to have an interpretation of types that relies on the calculus used, rather than on a mathematical model, like the bootstrap model. We then study the subtyping relation it induces and give our main contribution: the bootstrap model and the interpretation of types as sets of (pseudo-)values induce the same subtyping relation (Thm 1, § 5.1). The mathematical technicalities in the framework are not simple, but they are transparent to the final user. Thus, the overhead is hidden to the programmer. Incidentally, the full development of the theory of semantic subtyping is the main contribution of this paper w.r.t. the extended abstract [22]; in particular, § 3 was only sketched in *loc.cit.*, whereas here we provide full details and proofs of our results.

The benefits of our approach reside in that the programmer now has a language with an easy-to-write and very expressive set of types. Indeed, standard programming features in Java can be easily programmed in our framework (§ 8.1, § 8.3). Moreover, there are several benefits that make semantic subtyping very appealing in an object-oriented setting. For example, it allows us to easily handle powerful *boolean type constructors* (§ 2.1) and model both *structural* and *nominal* subtyping (§ 8.4).

The importance, both from the theoretical and the practical side, of boolean type constructors is widely known in several settings, e.g., in the λ-calculus. Below, we show two examples where the advantages of using boolean connectives in an object-oriented language become apparent.

Boolean constructors for modelling multimethods. Featherweight Java [21] is a core language, so several features of full Java are not included in it; in particular, an important missing feature is the possibility of overloading methods, both in the same class or along the class hierarchy. By using boolean constructors, the type of an overloaded method can be expressed in a very compact and elegant way, and this modeling comes for free after having defined the semantic subtyping machinery. Actually, what we are going to model is not Java's overloading (where the *static* type of the argument is considered for resolving method invocations) but *multimethods* (where the *dynamic* type is considered). To be precise, we implement the form of multimethods used, e.g., in [23, 24]; according to [25], this form of multimethods is "very clean and easy to understand [...] it would be the best solution for a brand new language".

As an example, consider the following class declarations:[1]

```
class A extends Object {                    class B extends A {
        ...                                         ...
        int length (string s){ ... }                int length (int n){ ... }
}                                           }
```

[1] Here and in the rest of the paper we use '...' to avoid writing the useless part of a class, e.g. constructors or irrelevant fields/methods.

As expected, method *length* of A has type $\mathbf{string} \to \mathbf{int}$. However, such a method in B has type $(\mathbf{string} \to \mathbf{int}) \wedge (\mathbf{int} \to \mathbf{int})$,[2] which can be simplified as $(\mathbf{string} \vee \mathbf{int}) \to \mathbf{int}$.

The use of negation types. Negation types are used by the compiler for typechecking terms of a language. They allow a clear definition of a type-case constructor, and more generally of pattern matching. But negation types could also be used directly by the programmer, even though their use is not of primary importance for programming usual tasks.

As an example, suppose we want to represent an inhabitant of Christiania, that does not want to use money and does not want to deal with anything that can be given a price. In this scenario, we have a collection of objects, some of which may have a *getValue* method that tells their value in €. We want to implement a class *Hippy* which has a method *barter* that is intended to be applied only to objects that do not have the method *getValue*. This is very difficult to represent in a language with only nominal subtyping; but also in a language with structural subtyping, it is not clear how to express the fact that a method is *not* present. In our framework we offer an elegant solution by assigning to objects that have the method *getValue* the type denoted by

$$[\mathit{getValue} : \mathbf{void} \to \mathbf{real}]$$

Within the class *Hippy*, we can now define a method with signature

$$\mathbf{void}\ \mathit{barter}(\neg[\mathit{getValue} : \mathbf{void} \to \mathbf{real}]\ x)$$

that takes in input only objects x that do not have a price, i.e., a method named *getValue*. One could argue that it is difficult to statically know that an object does not have method *getValue* and thus no reasonable application of method *barter* can be well typed. However, it is not difficult to explicitly build a collection of objects that do not have method *getValue*, by dynamically checking the presence of the method. This is possible thanks to the **instanceof** construction (described in § 8.3). Method *barter* can now be applied to any object of that list, and the application will be well typed.

In the case of a language with nominal subtyping, one can enforce the policy that objects with a price implement the interface *ValuedObject*. Then, the method *barter* would take as input only objects of type $\neg$*ValuedObject*.

While the example is quite simple, it should exemplify those situations in which we want to statically refer to a portion of a given class hierarchy and exclude the remainder. The approach we propose is more elegant and straightforward than possible solutions offered by current object-oriented paradigms.

Structural subtyping. An orthogonal issue, typical of object-oriented languages, is the *nominal* vs. *structural* subtyping question. In a language where subtyping is nominal, A is a subtype of B if and only if it is declared to be so, meaning if class A extends (or implements) class (or interface) B; these relations must be defined by the programmer and are based on the names of classes and interfaces declared. Java/C++/C# programmers are used to nominal subtyping, but other languages [26, 27, 28, 29, 30, 31, 32] are based on the structural approach. In the latter, subtyping relation is established only by analyzing the structure of a class, i.e. its fields and methods: the structural type of (the instances of) a class A is a subtype of the structural type

[2] To be precise, the actual type is $((\mathbf{string} \wedge \neg\mathbf{int}) \to \mathbf{int}) \wedge (\mathbf{int} \to \mathbf{int})$ but $\mathbf{string} \wedge \neg\mathbf{int} \simeq \mathbf{string}$, where $\simeq$ denotes $\leq \cap \leq^{-1}$ and $\leq$ is the (semantic) subtyping relation.

of (the instances of) a class B if and only if the fields and methods of A are a superset of the fields and methods of B, and their types in A are subtypes of their types in B (indeed, for fields we do not need type invariance because fields are immutable, since we are considering Featherweight Java that is functional). Even though the syntactic subtyping is more naturally linked to the nominal one, the former can also be adapted to support the structural one, as shown in [27, 29].

In this paper we follow the reverse direction and give another contribution. The definition of structural subtyping as set inclusion perfectly fits with the definition of semantic subtyping. However, with minor modifications, it is also possible to include in the framework the choice of using nominal subtyping without changing the underlying theory. It is not our aim to enter into the nominal vs. structural type question; both can be found in the literature (actually, structural typing is less used in real programming languages) and have their benefits. Thus, we take a neutral position on this aspect and allow them both in our setting; so, programmers can decide, from case to case, the kind of subtyping that better copes with their needs.

2 The calculus

In this section, we present the syntax of the calculus, starting with the types and then moving to the language terms, which are substantially the ones in FJ.

2.1 Types

Our types are defined by properly restricting the type terms inductively defined by the following grammar:

$$\begin{array}{lcll} \tau & ::= & \alpha \mid \mu & \textit{Type term} \\ \alpha & ::= & \mathbf{0} \mid \mathbb{B} \mid [\widetilde{l : \tau}] \mid \alpha \wedge \alpha \mid \neg\alpha & \textit{Object type } (\alpha\text{-type}) \\ \mu & ::= & \alpha \to \alpha \mid \mu \wedge \mu \mid \neg\mu & \textit{Method type } (\mu\text{-type}) \end{array}$$

Definition 1 (Types)**.** *The set of* types*, denoted by $\mathcal{T}$, is the largest set of well-formed regular trees produced by the syntax of type terms, where* regular *means that the tree has a finite number of non-isomorphic subtrees and* well-formed *means that the binary relation $\triangleright$ defined as*

$$\tau_1 \wedge \tau_2 \triangleright \tau_1 \qquad \tau_1 \wedge \tau_2 \triangleright \tau_2 \qquad \neg\tau \triangleright \tau$$

does not contain infinite chains.

Types can be of two kinds: α-types (used for declaring fields and, in particular, objects) and μ-types (used for declaring methods). Arrow types are needed to type the methods of our calculus. Since our language is first-order and methods are not first-class values, arrow types are introduced by a separate syntactic category, ranged over by μ. Indeed, even if later on we shall given an interpretation to arrow-types as sets of (pseudo-)values, this will be only a technical device to let our typing machinery work; no object will ever use methods as values (as customary in the OO paradigm).

Type $\mathbf{0}$ denotes the empty type. Type $\mathbb{B}$ denotes the basic types: integers, booleans, void, etc.[3]. Type $[\widetilde{l : \tau}]$ is a record type, denoted with ρ, where $\widetilde{l : \tau}$ indicates the sequence $l_1 : \tau_1, \ldots, l_k : \tau_k$, for some $k \geq 0$. Labels l range over an infinite countable

[3]The type **void** is different from $\mathbf{0}$ since the former is inhabited by only one value, whereas the latter is not inhabited by any any value.

set $\mathcal{L}$. When necessary, we will write a record type as $[\widetilde{a : \alpha}, \widetilde{m : \mu}]$ to emphasize the fields of the record, denoted by the labels $\widetilde{a}$, and the methods of the record, denoted by $\widetilde{m}$. Given a type $\rho = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\rho(a_i)$ is the type assigned to the field a_i and $\rho(m_j)$ is the type assigned to the method m_j. In each record type $a_i \neq a_j$ for $i \neq j$ and $m_h \neq m_k$ for $h \neq k$. To simplify the presentation, we are modeling a form of multimethods where at most one definition for every method name is present in every class. However, the general form of multimethods can be recovered by exploiting the simple encoding of § 8.2.

The boolean connectives $\wedge$ and $\neg$ have their intuitive set-theoretic meaning. We use $\mathbf{1}$ to denote the type $\neg\mathbf{0}$ that corresponds to the universal type. We use the abbreviation $\alpha\backslash\alpha'$ to denote $\alpha \wedge \neg\alpha'$ and $\alpha \vee \alpha'$ to denote $\neg(\neg\alpha \wedge \neg\alpha')$. The same holds for the μ-types.

Notice that every finite tree obtained by the grammar of types is both regular and well-formed; so, it is a type. Problems can arise with infinite trees; this leads us to restrict ourselves to the regular and the well-formed ones. Indeed, since we want our types to be usable in practice, we restrict ourselves to regular trees that can be easily written down in a finite way, e.g. by using recursive type equations. Moreover, as we want types to denote sets, we impose some restrictions to avoid ill-formed types. For example, the solution to $\alpha = \alpha \wedge \alpha$ contains no information about the set denoted by α; or $\alpha = \neg\alpha$ does not admit any solution. Such situations are problematic when we define the model. To rule them out, we only consider infinite trees whose branches always contain an atom, where *atoms* are the basic types $\mathbb{B}$, the record types $[\widetilde{l : \tau}]$ and the arrow types $\alpha \to \alpha$. This intuition is what the definition of relation $\triangleright$ formalizes. Since such a relation is strongly normalizing, it provides us with an induction principle on the set of types that we will use throughout the paper without any further reference to relation $\triangleright$. The restriction to well-formed types is required to avoid meaningless types; the same choice is used in [7] and in [33], where the same notion is called *contractiveness*.

In this paper, we express regular trees as the solution of recursive type definitions; this is in line with the aims of our work, that is lying down the theoretical basis for the system proposed. In a concrete implementation, we should replace this approach with a syntactic construct, like **rec** $X.\alpha$.

2.2 Terms

The syntax of our calculus is based on FJ [21] rather than, for example, the object-oriented calculus in [34], because of the widespread diffusion of Java. There is a correspondence between FJ and the pure functional fragment of Java, in a sense that any program in FJ is an executable program in Java. Our syntax is essentially the same as [21], apart from the absence of the *cast* construct, that we left out for the sake of simplicity: it is orthogonal to the aims of the current work and it can be added to the language without any major issue. Moreover, differently from FJ, we have complex types associated to field and method declarations, whereas in FJ everything is associated to class names. This is necessary because we deal with types resulting from the boolean combination of basic types. However, we shall show in §8.4 that complex types can be assigned symbolic names, to ease programming.

We assume a countable set of names, among which there are some key names: *Object* that indicates the root class, **this** that indicates the current object, and **super** that indicates the parent object. We will use the letters $A, B, C, \ldots$ for indicating classes, $a, b, \ldots$ for fields, $m, n, \ldots$ for methods and $x, y, z, \ldots$ for variables. $\mathcal{K}$ will denote the set of constants of the language and we will use the meta-variable c to range over $\mathcal{K}$.

Generally, to make examples clearer, we will use mnemonic names to indicate classes, methods, etc.; for example, *Point*, *print*, etc.

The syntax of the language is given by the following grammar:

Class declaration	$L ::= \textbf{class}\ C\ \textbf{extends}\ C\ \{\widetilde{\alpha\ a};\ K;\ \widetilde{M}\ \}$
Constructor	$K ::= C\ (\widetilde{\alpha\ x})\ \{\ \textbf{super}(\widetilde{x});\ \widetilde{\textbf{this}.a = x};\ \}$
Method declaration	$M ::= \alpha\ m\ (\alpha\ x)\ \{\ \textbf{return}\ e;\ \}$
Expressions	$e ::= x \mid c \mid e.a \mid e.m(e) \mid \textbf{new}\ C(\widetilde{e})$

A *program* is a pair $(\widetilde{L}, e)$ consisting of a sequence of class declarations $\widetilde{L}$, inducing a class hierarchy (as specified by the inheritance relation), and an expression e, that has to be evaluated therein. A class declaration L provides the name of the class, the name of the parent class it extends, its fields (each equipped with a type specification), the constructor K and its method declarations M. The constructor initializes the fields of the object by assigning values to the fields inherited by the super-class and to the fields declared in the present class. A method is declared by specifying the return type, the name of the method, the formal parameter (made up by a type specification given to a symbolic name) and a return expression, i.e. the body of the method. To increases readability of our theoretical development, we use unary methods. This does not compromise the expressivity of the language: passing tuples of arguments can be modeled by passing an object that instantiates a class, defined in an ad-hoc way by having as fields all the needed arguments. Finally, expressions e are variables, constants, field accesses, method invocations and object creations.

In this work we assume that $\widetilde{L}$ is well-defined, in the sense that "it is not possible that a class A extends a class B and class B extends class A", or "a constructor called B cannot be declared in a class A" and other obvious rules like these. All these kinds of checks could be carried out in the type system, but we prefer to assume them and focus our attention on the new features of our framework. The same sanity checks are assumed also in FJ [21].

To conclude, we want to remark that we could easily add the **rnd** construct of $\mathbb{C}$Duce to the syntax of our expressions. Precisely, $\textbf{rnd}(\alpha)$ returns a value of type α; this could be used to model nondeterministic methods and, in particular, user inputs. This addition would not change the theory we are going to develop.

3 Semantic Subtyping

Having defined the raw syntax, the next step is to introduce the typing rules, which typically involve a subsumption rule that uses a notion of subtyping. It is therefore necessary to define subtyping. As we have already said, in the semantic approach τ_1 is a subtype of τ_2 if all the τ_1-values are also τ_2-values, i.e., if the set of (well-typed) values of type τ_1 is a subset of the set of (well-typed) values of type τ_2. However, in this way, subtyping is defined by relying on the notion of well-typed values; hence, we need the typing relation to determine typing judgements for values; but the typing rules use the subtyping relation which we are going to define (in the so called "subsumption" rule). So, there is a circularity. To break this circle, we follow the path of [7] and adapt it to our framework.

The idea is to first interpret types as subsets of some abstract "model" and then establish subtyping as set-inclusion. By using this abstract notion of subtyping, we can then define the typing rules. Having now a notion of well-typed values, we can define the "real" interpretation of types as sets of values. This interpretation can be used to

define another notion of subtyping. But if the abstract model is chosen carefully, then the real subtyping relation coincides with the abstract one, and the circle is closed.

A model consists of a set D and an interpretation function $[\![\_]\!]_D : \mathcal{T} \to \mathcal{P}(D)$. Such a function should interpret boolean connectives in the expected way (conjunction corresponds to intersection and negation corresponds to complement – see Definition 2) and should capture the meaning of type constructors (see Definitions 3, 4 and 5). Notice that, given an intuitive meaning of types, there may be several models (defined in Definition 6) that satisfy this requirement, and it is not guaranteed that they all induce the same subtyping relation, nor that the induced subtyping is decidable. Aiming at a decidable subtyping, we shall only consider models for which values are finite trees (as defined in § 3.2).

Then, we only need one model that respects the intuitive meaning of type constructors and boolean connectives, and whose values are finite trees. We shall construct such a model in § 3.3 and call it the *bootstrap model*. Then, set inclusion in the bootstrap model induces a (decidable) subtyping relation, $\tau_1 \leq_{\mathcal{B}} \tau_2 \iff [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$, to be used for typing terms and breaking circularity.

3.1 Set-theoretic interpretations and models

First of all, any set-theoretic interpretation must respect the set-theoretic meaning of the boolean constructors; this is formalized in the following definition.

Definition 2 (Set-theoretic interpretation)**.** *A* set-theoretic interpretation *of types in $\mathcal{T}$ is given by a set D and a function $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ such that, for any $\tau_1, \tau_2, \tau \in \mathcal{T}$, the following hold:*

$$[\![\mathbf{0}]\!] = \varnothing \qquad [\![\tau_1 \wedge \tau_2]\!] = [\![\tau_1]\!] \cap [\![\tau_2]\!] \qquad [\![\neg\tau]\!] = D \setminus [\![\tau]\!]$$

Notice that the above definition implies $[\![\tau_1 \vee \tau_2]\!] = [\![\tau_1]\!] \cup [\![\tau_2]\!]$, $[\![\tau_1 \setminus \tau_2]\!] = [\![\tau_1]\!] \setminus [\![\tau_2]\!]$ and $[\![\mathbf{1}]\!] = D$. Every set-theoretic interpretation $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ induces a binary relation $\leq_{[\![]\!]} \subseteq \mathcal{T}^2$ defined as follows: $\tau_1 \leq_{[\![]\!]} \tau_2 \iff [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. This relation is the semantic subtyping relation. Thanks to negation, the problem of deciding the subtyping between two types is reduced to the problem of identifying the empty sets, that is: $[\![\tau_1]\!] \subseteq [\![\tau_2]\!] \iff [\![\tau_1]\!] \setminus [\![\tau_2]\!] = \varnothing \iff [\![\tau_1]\!] \cap (D \setminus [\![\tau_2]\!]) = \varnothing \iff [\![\tau_1 \wedge \neg\tau_2]\!] = \varnothing$.

Next, we are going to define the requirements for a set-theoretic interpretation to correctly represent the meaning of the type constructors. First, we require that, for every basic type $\mathbb{B}$, there is a set of values $Val_{\mathbb{B}}$ ($\subseteq \mathcal{K}$) for that type. Conversely, we also require that every sets of basic values correspond to a basic type. In particular, for every constant c, there is a basic type $\mathbb{B}_c$ such that $Val_{\mathbb{B}_c} = \{c\}$; so, the set of constants that inhabit the basic type $\mathbb{B}_c$ is composed only by the singleton constant c. For example, $\mathbb{B}_1$ is the type that contains only 1 as value. By using union, we can then combine these types to obtain fine-grained type specification: e.g., $\mathbb{B}_1 \vee \mathbb{B}_2 \vee \mathbb{B}_3$ can be used to declare that a field can only be assigned values 1, 2 or 3.

For a record type $\rho = [\widetilde{l : \tau}]$, the intuition is that it should represent all objects that, in the field l_i, have values of type τ_i, and that may have other fields as well. Formally it should be the set of relations $R \subseteq \mathcal{L} \times D$ such that $\forall d \in D\ \forall i\ .((l_i, d) \in R \Rightarrow d \in [\![\tau_i]\!])$. Also, the record type $[a : \mathbf{0}]$ should be interpreted as the empty set, as our intuition suggests that no object of this type can be instantiated. Thus we add the requirement that $dom(R) \supseteq \{\widetilde{l}\}$.

Definition 3. *Given a record type $[\widetilde{l : \tau}]$, we define $[\![[\widetilde{l : \tau}]]\!]$ as:*

$$[\![[\widetilde{l : \tau}]]\!] = \{R \subseteq (\mathcal{L} \times D) \mid dom(R) \supseteq \{\widetilde{l}\} \ \wedge\ \forall d \in D\ \forall i\ .((l_i, d) \in R \Rightarrow d \in [\![\tau_i]\!])\}$$

Now we move to arrow types. For a type $\alpha_1 \to \alpha_2$, the intuition is that it should represent the set of functions f such that $\forall d \in D.(d \in [\![\alpha_1]\!] \Rightarrow f(d) \in [\![\alpha_2]\!])$. We consider binary relations instead of functions because this simplifies the equations satisfied by the types and can be used also to model nondeterministic methods (this is similar to [7]); such a feature would arise e.g. in an extension of the language with side effects. Before defining the interpretation of an arrow type, we introduce the notion of type error: it is not possible to invoke an arbitrary method on an arbitrary argument. Said differently, this notion is used to avoid that the invocation of any well-typed method on any well-typed argument, is itself well-typed. To assure this, we will use Ω as a special element to denote this type error. So, we will interpret a type $\alpha_1 \to \alpha_2$ as the set of binary relations $Q \subseteq D \times D_\Omega$ (where $D_\Omega = D \uplus \{\Omega\}$), such that $\forall (q, q') \in Q.(q \in [\![\alpha_1]\!] \Rightarrow q' \in [\![\alpha_2]\!])$.

Definition 4. *Let D be a set and X, Y subsets of D, then we define:*

$$X \to Y = \{Q \subseteq D \times D_\Omega \mid \forall (q, q') \in Q.(q \in X \Rightarrow q' \in Y)\}$$

Let us now show how Ω is needed to model this kind of type error. If we replace D_Ω with D in the definition above, then $X \to Y$ would always be a subset of $D \to D$. By applying Definition 4, $D \to D$ is interpreted as the set of relations $Q_D \subseteq D \times D$ such that $\forall (q, q') \in Q_D.(q \in D \Rightarrow q' \in D)$. For every relation Q of $X \to Y$ with $X, Y \subseteq D$, it is easy to see that for all pairs $(q, q') \in Q$, it holds that $q \in X \subseteq D \Rightarrow q' \in Y \subseteq D$. Namely, every pair in Q belongs also to Q_D. This would imply that any arrow type would be a subtype of $\mathbf{1} \to \mathbf{1}$. Then, by using the subsumption rule, the invocation of any well-typed method on any well-typed argument would be well-typed, violating the type-safety property of the calculus. With the definition given above, we have $X \to Y \subseteq D \to D$ if and only if $D \subseteq X$, because of contra-variance of arrow types.

At this point, we can give the formal definition of an extensional interpretation associated with a set-theoretic interpretation.

Definition 5 (Extensional interpretation)**.** *Let $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation of $\mathcal{T}$. We define its associated* extensional interpretation *as the set-theoretic interpretation $\mathbb{E}(\_) : \mathcal{T} \to \mathcal{P}(\mathbb{E}D)$ (where $\mathbb{E}D = \mathcal{K} \uplus \mathcal{P}(\mathcal{L} \times D) \uplus \mathcal{P}(D \times D_\Omega)$) such that:*

$$\begin{array}{rll} \mathbb{E}(\mathbb{B}) = & Val_{\mathbb{B}} & \subseteq \mathcal{K} \\ \mathbb{E}\left(\widetilde{[l : \tau]}\right) = & [\![\widetilde{[l : \tau]}]\!] & \subseteq \mathcal{P}(\mathcal{L} \times D) \\ \mathbb{E}(\alpha_1 \to \alpha_2) = & [\![\alpha_1]\!] \to [\![\alpha_2]\!] & \subseteq \mathcal{P}(D \times D_\Omega) \end{array}$$

For a set-theoretic interpretation $[\![\cdot]\!]$ to be a model, we will require it to behave the same way as its extensional interpretation, as far as subtyping is concerned.

Definition 6 (Model)**.** *A set-theoretic interpretation $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ is a* model *if it induces the same subtyping relation as its associated extensional interpretation:*

$$\forall \tau_1, \tau_2 \in \mathcal{T}. \quad [\![\tau_1]\!] \subseteq [\![\tau_2]\!] \iff \mathbb{E}(\tau_1) \subseteq \mathbb{E}(\tau_2)$$

The observation we have done before on the problem of emptiness permits us to write the condition on types given in the definition of model as:

$$\forall \tau \in \mathcal{T}. \, [\![\tau]\!] = \varnothing \iff \mathbb{E}(\tau) = \varnothing$$

3.2 Well-founded model

Among all possible models, we focus our attention to those that capture a very important property, namely that values are finite m-ary trees whose leaves are constants. For example, let us consider the recursive type $\alpha = [a : \alpha]$. Intuitively, a value u has this type if and only if it is an object **new** $C(u')$, where u' has also type α; hence, u' should be of the form **new** $C(u'')$, where u'' has again type α; and so on. Thus, such a value would be an infinite tree **new** $C($**new** $C($**new** $C(\cdots)))$, that is excluded since values are the result of some non diverging computation based on the strict call-by-value evaluation strategy. Furthermore, notice that the syntax of our calculus rules out a class declaration like

class C **extends** B {
    $C\ a$;
    $C()$ {**this**.a = **this**;}
}

In particular, the form of constructors that we assume (the same as in [21]) requires a value for every field of the class (see the comment to rule *(class)* from Table 1 later on) and this is not the case here. Finally, since we are working with a functional fragment of Java, no assignment will eventually assign **this** to a. As a consequence, values cannot be cyclic (in the sense that they have pointers to themselves) and the type $\alpha = [a : \alpha]$ cannot not contain values. Clearly this does not imply that all recursive types are trivial. In § 8.1 we will see that it is possible to satisfy the property we have just introduced and still use recursive types, e.g., to create lists.

We now formalize the intuition above, i.e. that values are finite m-ary trees whose leaves are constants.

Definition 7 (Structural interpretation)**.** *A set-theoretic interpretation* $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ *is* structural *if:*

- $\mathcal{P}_f(\mathcal{L} \times D) \subseteq D$*, where* $\mathcal{P}_f(\cdot)$ *denotes the finite powerset;*
- *for any* $\widetilde{\tau}$*, it holds that* $[\![\widetilde{[l : \tau]}]\!] \subseteq \mathcal{P}_f(\mathcal{L} \times D)$*;*
- *the binary relation* $\gg$ *on* $\mathcal{P}_f(\mathcal{L} \times D) \times D$ *defined as* $\{(l_1, d_1), \ldots, (l_n, d_n)\} \gg d_i$*, for* $i \in \{1, \ldots, n\}$*, does not admit infinite descending chains.*

Definition 8 (Well-founded model)**.** *A model* $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ *is* well-founded *if it induces the same subtyping relation as a structural set-theoretic interpretation.*

3.3 Bootstrap model

Now that we have all the necessary ingredients, we have to show that a well-founded model exists, and use it as the bootstrap model. In order to achieve this, we need some preliminary notions, which will also be used later on in our formal development. As already mentioned earlier, there are three kinds of atomic types: basic types (**basic**), record types (**rec**) and functional types (**fun**). We use $\mathbb{T}_{\mathbf{basic}}$, $\mathbb{T}_{\mathbf{rec}}$ and $\mathbb{T}_{\mathbf{fun}}$ for basic, record and functional types, respectively. We use $\mathbb{T}$ to indicate the set of atomic types and we let t range over this set. Then, $\mathbb{T} = \mathbb{T}_{\mathbf{basic}} \uplus \mathbb{T}_{\mathbf{rec}} \uplus \mathbb{T}_{\mathbf{fun}}$.

Definition 9 (Finitely extensional interpretation)**.** *A set-theoretic interpretation* $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ *is* finitely extensional *if the following hold:*

- $D = \mathbb{E}_f D$

- $[\![t]\!] = \mathbb{E}(t) \cap D$, *for every atomic type* t.

where, for every set D, *we let* $\mathbb{E}_f D$ *be* $\mathcal{K} \uplus \mathcal{P}_f(\mathcal{L} \times D) \uplus \mathcal{P}_f(D \times D_\Omega)$ *and* $\mathcal{P}_f$ *is the powerset of finite subsets.*

By a simple induction on types, it is easy to prove that, in every finitely extensional interpretation $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$, it holds that $[\![\tau]\!] = \mathbb{E}(\tau) \cap D$, for every type $\tau \in \mathcal{T}$. The advantage of the above definition is that it easily permits us to construct a model, by exploiting the following lemma.

Lemma 1. *Every finitely extensional interpretation is a model.*

Proof (Sketch). The proof follows the same lines as [7]. In order to show that a finitely extensional interpretation is a model we need to identify the empty sets of a set-theoretic interpretation and its associated extensional one. Namely, $\mathbb{E}(\tau) = \varnothing \iff \mathbb{E}(\tau) \cap D = \varnothing$. The proof is completed by using a series of equivalences starting from the predicate $\mathbb{E}(\tau) \cap D = \varnothing$. The details of such equivalences are the same as in [7]. □

We are now ready to construct a structural and finitely extensional interpretation, which will be used in the remainder of our work as the bootstrap model. To construct such a model, let us define a set $\mathcal{B}$ such that $\mathcal{B} = \mathbb{E}_f \mathcal{B}$. This means that $\mathcal{B}$ is the solution of the equation $\mathcal{B} = \mathcal{K} \uplus \mathcal{P}_f(\mathcal{L} \times \mathcal{B}) \uplus \mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega)$, where again $\mathcal{P}_f$ denotes the powerset of finite subsets. Hence, by construction, it turns out that $\mathcal{B}$ is the set of finite terms generated by the following grammar:

$$d ::= c \mid \{(l, d), \cdots, (l, d)\} \mid \{(d, d'), \cdots, (d, d')\} \qquad\qquad d' ::= d \mid \Omega$$

We now define a set-theoretic interpretation

$$[\![\tau]\!]_{\mathcal{B}} = \{d \in \mathcal{B} \mid d : \tau\}$$

Here, judgement $d' : \tau$ is inductively defined on the lexicographic order of the pairs $\langle d', \tau \rangle$, by exploiting the inductive structure of the elements of $\mathcal{B}$ and the induction principle for types. Its defining clauses are the following ones (it is assumed to be false in every non-depicted case):

$$\begin{array}{rll}
c : \mathbb{B} & \text{iff} & c \in Val_{\mathbb{B}} \\
\{(l_1, d_1), \cdots, (l_n, d_n)\} : [l'_1 : \tau_1, \cdots, l'_m : \tau_n] & \text{iff} & \{l'_1, \cdots, l'_m\} \subseteq \{l_1, \cdots, l_n\} \text{ and} \\
 & & \forall i, j. (l_i = l'_j \Rightarrow d_i : \tau_j) \\
\{(d_1, d'_1), \cdots, (d_n, d'_n)\} : \alpha \to \alpha' & \text{iff} & \forall i.\ (d_i : \alpha \Rightarrow d'_i : \alpha') \\
d : \tau_1 \wedge \tau_2 & \text{iff} & d : \tau_1 \text{ and } d : \tau_2 \\
d : \neg\tau & \text{iff} & \text{not } d : \tau
\end{array}$$

Notice that this induction is well-founded since d is finite and τ is well-formed.

Proposition 1. $[\![\_]\!]_{\mathcal{B}}$ *is a structural and finitely extensional interpretation; thus, it is a well-founded model.*

Proof. The set-theoretic interpretation $[\![\tau]\!]_{\mathcal{B}} = \{d \in \mathcal{B} \mid d : \tau\}$ is constructed in a way such that $[\![\tau]\!] = \mathbb{E}(\tau) \cap \mathcal{B}$; this fact, together with $\mathcal{B} = \mathbb{E}_f \mathcal{B}$, entails that it is finitely extensional (see Definition 9). Hence, by Lemma 1 it is a model. Moreover, it is easy to see that it is structural, hence it is a well-founded model (see Definition 8). □

4 The Type System

The type system for our language uses the subyping relation just defined to derive typing judgements of the form $\Gamma \vdash_{\mathcal{B}} e : \tau$. In particular, this means to use $\leq_{\mathcal{B}}$ in the subsumption rule. In the following we just write $\leq$ instead of $\leq_{\mathcal{B}}$.

Let us assume a sequence of class declarations $\widetilde{L}$. First of all, we have to determine the (structural) type of every class C in $\widetilde{L}$. To this aim, we have to take into account the inheritance relation specified in the class declarations in $\widetilde{L}$. We write "$a \in C$" to mean that there is a field declaration of name a in class C within the hierarchy $\widetilde{L}$. Similarly, we write "$a \in C$ with type α" to also specify the declared type α. Similar notations also hold for method names m.

Definition 10.

- *type*$(Object) = [\,]$;
- *type*$(C) = \rho$, *provided that:*
  - C *extends* D *in* $\widetilde{L}$;
  - *type*$(D) = \rho'$;
  - *for any field name* a
    - *if* $\rho'(a)$ *is undefined and* $a \notin C$, *then* $\rho(a)$ *is undefined;*
    - *if* $\rho'(a)$ *is undefined and* $a \in C$ *with type* α'', *then* $\rho(a) = \alpha''$;
    - *if* $\rho'(a)$ *is defined and* $a \notin C$, *then* $\rho(a) = \rho'(a)$;
    - *if* $\rho'(a)$ *is defined,* $a \in C$ *with type* α'' *and* $\alpha'' \leq \rho'(a)$, *then* $\rho(a) = \alpha''$.

    We assume that all the fields defined in ρ' *and not declared in* C *appear at the beginning of* ρ, *having the same order as in* ρ'; *the fields declared in* C *then follow, respecting their declaration order in* C.
  - *for any method name* m:
    - *if* $\rho'(m)$ *is undefined and* $m \notin C$, *then* $\rho(m)$ *is undefined;*
    - *if* $\rho'(m)$ *is undefined and* $m \in C$ *with type* $\alpha \to \alpha'$, *then* $\rho(m) = \alpha \to \alpha'$;
    - *if* $\rho'(m)$ *is defined and* $m \notin C$, *then* $\rho(m) = \rho'(m)$;
    - *if* $\rho'(m) = \bigwedge_{i=1}^{n} \alpha_i \to \alpha_i'$, $m \in C$ *with type* $\alpha \to \alpha'$ *and* $\mu = \alpha \to \alpha' \wedge \bigwedge_{i=1}^{n}((\alpha_i \setminus \alpha) \to \alpha_i') \leq \rho'(m)$, *then* $\rho(m) = \mu$.
- *type*(C) *is undefined, otherwise.*

Definition 10 inductively defines the partial function *type*(C) on the class hierarchy $\widetilde{L}$ (of course this induction is well founded since $\widetilde{L}$ is finite); when defined, it returns a record type ρ. In particular, the type of a method is a boolean combination of arrow types declared in the current and in the parent classes. This follows the same lines as [7] in order to deal with habitability of types. The condition $\rho(m) \leq \rho'(m)$ imposed in the method declaration is mandatory to assure that the type of C is a subtype of the type of D; without such a condition, it would be possible to have a class whose type is not a subtype of the parent class. If it were the case, type soundness would fail, as the following example shows.

Typing Expressions :

$$(subsum)\ \frac{\Gamma \vdash e : \alpha_1 \quad \alpha_1 \leq_{\mathcal{B}} \alpha_2}{\Gamma \vdash e : \alpha_2} \qquad (const)\ \frac{}{\Gamma \vdash c : \mathbb{B}_c} \qquad (var)\ \frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha}$$

$$(field)\ \frac{\Gamma \vdash e : [a : \alpha]}{\Gamma \vdash e.a : \alpha} \qquad (m\text{-}inv)\ \frac{\Gamma \vdash e_2 : [m : \alpha_1 \to \alpha_2] \quad \Gamma \vdash e_1 : \alpha_1}{\Gamma \vdash e_2.m(e_1) : \alpha_2}$$

$$(new)\ \frac{\begin{array}{c} type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}] \qquad \Gamma \vdash \widetilde{e} : \widetilde{\alpha'} \qquad \widetilde{\alpha'} \leq_{\mathcal{B}} \widetilde{\alpha} \\ \alpha' = [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \wedge \bigwedge_{i \in I} \neg[\widetilde{l_i : \tau_i}] \not\simeq \mathbf{0} \end{array}}{\Gamma \vdash \mathbf{new}\ C(\widetilde{e}) : \alpha'}$$

Typing Method Declarations :

$$(m\text{-}decl)\ \frac{x : \alpha_1, \mathbf{this} : type(C) \vdash e : \alpha_2}{\vdash_C \alpha_2\ m\ (\alpha_1\ x)\{\mathbf{return}\ \ e\}}$$

Typing Class Declarations :

$$(class)\ \frac{type(D) = [\widetilde{a' : \alpha'}, \widetilde{m : \mu}] \quad K = C(\widetilde{\alpha'\ x'}, \widetilde{\alpha\ x})\{\mathbf{super}(\widetilde{x'}); \widetilde{\mathbf{this}.a = x}\} \quad \vdash_C \widetilde{M}}{\vdash \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\widetilde{\alpha\ a}\ K\ \widetilde{M}\}}$$

Typing Programs :

$$(prog)\ \frac{\vdash \widetilde{L} \qquad \vdash e : \alpha}{\vdash (\widetilde{L}, e)}$$

Table 1: Typing rules

```
class C extends Object {
    ...
    real m(real x) {return x}
    real F() {return this.m(3)}
}
```

```
class D extends C {
    ...
    compl m(int x) {return x × √−1}
    real G() {return this.F()}
}
```

As usual **int** ≤ **real** ≤ **compl**. At run time, the function G returns a complex number, instead of a real one. The example shows that, when the method m is overloaded, we have to be sure that the return type is a subtype of the original type. Otherwise, due to the dynamic instantiation of **this**, there may be a type error. A similar argument justifies the condition $\alpha'' \leq \rho'(a)$ imposed for calculating function *type* for field names.

Let us now consider the typing rules given in Table 1. We assume Γ to be a typing environment, i.e., a finite sequence of α-type assignments to variables. Most rules are very intuitive. Rule (*subsum*) permits to derive for an expression e of type α_1 also a type α_2, if α_1 is a subtype of α_2. Notice that, for the moment, the subtyping relation used in this rule is the one induced by the bootstrap model. In rule (*const*), we assign type $\mathbb{B}_c$ to the constant c; if c belongs to a larger basic type $\mathbb{B}$, then by subsumption we can also assign to c such a larger type. Rule (*var*) derives that x has type α, if x is

assigned type α in Γ.

Rule (*field*) states that, if an expression e has type $[a : \alpha]$, we can access the field a of e and the type of the expression $e.a$ is α. Rule (*m-inv*) states that, if an expression e_2 is of type $[m : \alpha_1 \to \alpha_2]$ and an expression e_1 is of type α_1, we can invoke method m of e_2 with argument e_1 and the type of the expression $e_2.m(e_1)$ is α_2. Notice that in these two rules the record types are singletons, as it is enough that inside the record type there is just the field or the method that we want to access or invoke. If the record type is more specific (having other fields or methods), we can still get the singleton record by using the subsumption rule. The rules *m-inv* models methods as invariant in their arguments. This is not restrictive, as we can always use subsumption to promote the type of the argument to match the declared type of the method.

For rule (*new*), an object creation can be typed by recording the actual type of the arguments passed to the constructor, since we are confining ourselves to the functional fragment of the language. Of course, if we move to the setting where fields can be modified, it is unsound to record the actual type of the initial values, since during the computation a field could be updated with values of its declared type. Moreover, like in [7], we can extend the type of the object by adding any record type that cannot be assigned to it, as long as this does not lead to a contradiction (i.e., a type semantically equivalent to $\mathbf{0}$). This possibility of adding negative record types is not really necessary for programming purposes: it is only needed to ensure that every non-zero type has at least one value of that type. In particular, we want that the union of a type τ and its negation $\neg\tau$ gives $\mathbf{1}$. For this to be the case we want that the union of the interpretation of τ and the interpretation of $\neg\tau$ gives $\mathcal{B}$. By adding the negative record types, rule (*new*) permits us to have typing derivations also for types of the form $\neg\tau$. This property guarantees that the interpretation of types as sets of values induces the same subtyping relation as the bootstrap model and will be used when proving that the interpretation of types as sets of values is a set-theoretic interpretation.

Finally, rule (*m-decl*) checks when a method declaration is acceptable for a class C; this can only happen if $type(C)$ is defined. Rules (*class*) and (*prog*) check when a class declaration and a program are well-typed and are similar to the ones in FJ. In particular, notice that rule (*class*) imposes that every field declared in the class and every field inherited from the super-class has an assignment in the constructor: this comes from the fact that the sequences of types $\widetilde{\alpha'}$ and $\widetilde{\alpha}$ in the constructor declaration are those occurring in the type of the super-class and in the class declaration, respectively.

Similarly to [7, 10], the type checking relation is decidable.

5 Types as Sets of (Pseudo-)Values

Having defined the type system for our language, we are now ready to give an interpretation of types as sets of values. To close the circle, the first thing to do is to define values in our calculus. As usual, values are the results of (well-typed) computations, given by a small step operational semantics that we are going to introduce in the next section; so, values are those expressions that cannot be reduced further. Formally, they are produced by the following grammar:

$$u ::= c \mid \mathbf{new}\ C(\widetilde{u})$$

that is, values are constants or objects initialized by passing values to their constructor.

However, as the classes in $\widetilde{L}$ are finite, with these values we are able to inhabit just a finite number of record types. Also, since we have not higher-order values, the μ-types would not be inhabited. This is a major technical difference w.r.t. [7]. In order to

overcome this issue, we define *pseudo-values*, that are only used for interpreting types and deciding the subtyping relation.

In order to define the set of pseudo-values in our calculus, we want to be independent of the classes declared by a programmer in $\widetilde{L}$. Hence, we assume that, for every well-formed record type ρ, there exists a class name C_ρ such that $type(C_\rho)$ is defined to be ρ. It is important to also notice that in this way we inhabit only the "well-defined" record types, that is only those that can instantiate (and create an object of) a class corresponding to the record we are dealing with. For example, $\mathbf{new}\ C_{[a:\mathbf{0}]}(\cdot)$ does not create any object, as no value of type $\mathbf{0}$ exists (thus, it is impossible to instantiate a class of type $[a : \mathbf{0}]$).

In order to inhabit method types, we add to the syntax of values an abstraction construct $\lambda_{(m,\rho)}x.e$ that intuitively specifies the argument x and the body e of a method m in a record ρ (that will be inhabited through class C_ρ).

Formally, pseudo-values are produced by the following grammar:

$$v ::= u \mid \lambda_{(m,\rho)}x.e$$

Since abstractions are not part of our original syntax, we also need to add the following typing rule for them:

$$\frac{\mu = \bigwedge_{i\in I}(\alpha_i \to \alpha'_i) \wedge \bigwedge_{j\in J} \neg\left(\hat{\alpha}_j \to \hat{\alpha}'_j\right) \not\simeq \mathbf{0} \qquad \rho(m) = \bigwedge_{i\in I}(\alpha_i \to \alpha'_i) \qquad \forall i \in I\,.\,\Gamma, x : \alpha_i \vdash e : \alpha'_i}{\Gamma \vdash \lambda_{(m,\rho)}x.e : \mu}\ (abstr)$$

Rule (*abstr*) states that an abstraction $\lambda_{(m,\rho)}x.e$ related to method m in the class typed by ρ is of type μ, with $\mu = \bigwedge_{i\in I}(\alpha_i \to \alpha'_i) \wedge \bigwedge_{j\in J} \neg\left(\hat{\alpha}_j \to \hat{\alpha}'_j\right) \not\simeq \mathbf{0}$. The positive part of the conjunction in μ comes from the "real" types of method m given by the function $type()$, whereas the negative part of the conjunction follows the same intuition as the rule (*new*) given in the previous section.

It is worth noting that not all values and pseudo-values can be typed. For example, there exists no type that can be assigned to $\mathbf{new}\ C_{[x:\mathbf{int}]}()$, $\mathbf{new}\ C_{[x:\mathbf{int}]}(1, 2)$, $\mathbf{new}\ C_{[x:\mathbf{int}]}(\text{“}foo\text{”})$, where $C_{[x:\mathbf{int}]}$ is the name of a class with a field x of type $\mathbf{int}$; a similar situation arises for $\lambda_{(m,[m:\alpha\to\alpha])}x.y$, $\lambda_{(m,[\,])}x.x$ and $\lambda_{(m,[m:\mathbf{int}\to\mathbf{string}])}x.x$. Hence, in what follows, we shall only consider typeable values and pseudo-values; in particular, let $\mathcal{V}$ denote the set of typeable pseudo-values. Then, the interpretation of a type τ as a set of pseudo-values is

$$[\![\tau]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : \tau\}$$

From now on, we shall simply call pseudo-values the typeable pseudo-values.

5.1 Closing the circle

As we already discussed, the bootstrap model $[\![\cdot]\!]_{\mathcal{B}}$ induces the following subtyping relation:

$$\tau_1 \leq_{\mathcal{B}} \tau_2 \iff [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$$

In the typing rules for our language, we used $\leq_{\mathcal{B}}$ to derive typing judgements of the form $\Gamma \vdash_{\mathcal{B}} e : \tau$. Similarly, the typing judgements for the language allow us to define a new natural set-theoretic interpretation of types, the one based on (pseudo-)values $[\![\tau]\!]_{\mathcal{V}}$, and then a new ("real") subtyping relation:

$$\tau_1 \leq_{\mathcal{V}} \tau_2 \iff [\![\tau_1]\!]_{\mathcal{V}} \subseteq [\![\tau_2]\!]_{\mathcal{V}}$$

$$\textit{(f-ax)}\ \frac{type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]}{(\textbf{new}\ C(\widetilde{u})).a_i \to u_i} \qquad \textit{(m-ax)}\ \frac{body(m, u, C) = \lambda x.e}{(\textbf{new}\ C(\widetilde{u})).m(u) \to e[{}^{u}/_{x}, {}^{\textbf{new}\ C(\widetilde{u})}/_{\textbf{this}}]}$$

$$\textit{(m-red}_1\textit{)}\ \frac{e' \to e''}{e'.m(e) \to e''.m(e)} \qquad \textit{(m-red}_2\textit{)}\ \frac{e' \to e''}{e.m(e') \to e.m(e'')} \qquad \textit{(f-red)}\ \frac{e \to e'}{e.a \to e'.a}$$

$$\textit{(n-red)}\ \frac{e_i \to e_i'}{\textbf{new}\ C(e_1, \ldots, e_i, \ldots, e_k) \to \textbf{new}\ C(e_1, \ldots, e_i', \ldots, e_k)}$$

Table 2: Operational semantics

The new relation $\leq_{\mathcal{V}}$ might in principle be different from $\leq_{\mathcal{B}}$. However, since the definitions of the model, of the language and of the typing rules have been carefully chosen, the two subtyping relations coincide, as Theorem 1 shows. Because of this result, from now on we shall be sloppy and avoid the subscripts $\mathcal{B}$ and $\mathcal{V}$ in $\vdash_{\mathcal{B}}$, $\leq_{\mathcal{B}}$ and $\leq_{\mathcal{V}}$; we shall simply write $\vdash$ and $\leq$.

Theorem 1. *The bootstrap model $[\![\cdot]\!]_{\mathcal{B}}$ induces the same subtyping relation as $[\![\cdot]\!]_{\mathcal{V}}$.*

The proof of this theorem is the main technical challenge in every paper on semantic subtyping. It requires a lot of work that, partly, mimics what is done, e.g., in [7]. All details adapted to our framework are in the Appendix.

6 The Operational Semantics

The operational semantics for our language is defined through the reduction rules in Table 2; these are essentially the same as in FJ. There are only two notable differences: we do not need to define an ad-hoc function to extract the fields of an object, but we simply use function *type* already defined; function *body* also depends on the (type of the) method argument, necessary for finding the appropriate declaration when we have multimethods.

We fix the set of class declarations $\widetilde{L}$ and define the operational semantics as a binary relation on the expressions of the calculus $e \to e'$, called *reduction relation*. The axiom for field access *(f-ax)* states that, if we try to access the i-th field of an object, we just return the i-th argument passed to the constructor of that object. We have used the premise $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ as we want all the fields of the object instantiating class C: function $type(C)$ provides them in the right order (i.e., the order in which the constructor of class C expects them to be). The axiom for method invocation *(m-ax)* tries to match the argument of a method in the current class and, if a proper type match is not found, it looks up in the hierarchy; these tasks are carried out by function *body*, whose definition is the following (the cases must be considered in order):

$$body(m, u, C) = \begin{cases} \lambda x.e & \text{if } C \text{ contains } \alpha' m(\alpha\ x)\{\textbf{return}\ e\} \text{ and } \vdash u : \alpha, \\ body(m, u, D) & \text{if } C \text{ extends } D \text{ in } \widetilde{L}, \\ UNDEF & \text{otherwise.} \end{cases}$$

Function $body(m, u, C)$ controls whether method m is declared in class C and if argument u passed to m has the appropriate type, viz. the type of the formal argument of the method. Otherwise, the parent class of C is checked. If method look-up does not give

any result, function $body(m, u, C)$ is not defined. Notice that method resolution is performed at runtime, by keeping into account the *dynamic* type of the argument; hence, *multimethods* are supported, differently from what happens in Java, where overloading resolution is performed at compile time by keeping into account the *static* type of the argument. We choose the first way because, in our view, is more intuitive. A more traditional modelling of overloading is possible and easy to model.

Moreover, as already noted in § 2.1, we use simplified multimethods, where at most one declaration for every method name is present in every class. This simplifies the definition of functions *body* and *type*. However, richer forms of multimethods can be assumed in our framework, at the price of complicating the definitions of such functions. In particular, function *body* can be rendered in the general setting by following [35]. In our view, a better alternative is the encoding of the more general setting provided in § 8.2.

To complete the definition of the operational semantics, we need the structural rules *(f-red)*, *(m-red$_1$)*, *(m-red$_2$)* and *(n-red)*, to transform the target of a method invocation or of a field access into a value.

7 Soundness of the Type System

Theorem 1 does not automatically imply that the definitions put forward in § 3, 4 and 5 are "valid" in any formal sense, only that they are mutually coherent. To complete the theoretical treatment, we need to check type soundness, stated by the following theorems. We proceed in the standard way, by stating the theorems of *subject reduction* and *progress*, that can be proved by exploiting a few auxiliary lemmata.

Lemma 2. *If D is an ancestor of C in the class hierarchy, then $type(C) \leq type(D)$.*

Proof. By induction on the distance between C and D. The base case is trivial. For the inductive case, we have that C extends C' that has D as one of its ancestors. By inductive hypothesis, $type(C') \leq type(D)$; thus, it suffices to prove that $type(C) \leq type(C')$. This easily follows from the definition of function *type*. □

Lemma 3 (Strengthening)**.** *Let Γ_1 and Γ_2 by type environments, such that $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ and, for every $x \in dom(\Gamma_1) : \Gamma_2(x) \leq \Gamma_1(x)$. If $\Gamma_1 \vdash e : \alpha$ then $\Gamma_2 \vdash e : \alpha$.*

Proof. By induction on the derivation tree for $\Gamma_1 \vdash e : \alpha$. □

Lemma 4 (Substitution)**.** *If $\Gamma \vdash e : \alpha$, with $\Gamma = \Gamma', x : \alpha'$ and $\Gamma' \vdash e' : \alpha'$, then $\Gamma' \vdash e[^{e'}/_x] : \alpha$.*

Proof. By induction on the structure of e. For the base case, let us first assume that $e = x$; in this case, the thesis trivially follows by the facts that $\alpha = \alpha'$ and $e[^{e'}/_x] = e'$. On the contrary, if $e \neq x$, then $e[^{e'}/_x] = e$ and the thesis holds. For the inductive step, let us consider the possible forms of e:

$e = e''.a$**:** by the typing rule for field access, we have that $\Gamma \vdash e'' : [a : \alpha]$. By inductive hypothesis, $\Gamma' \vdash e''[^{e'}/_x] : [a : \alpha]$ and hence $\Gamma' \vdash (e''.a)[^{e'}/_x] : \alpha$, since $a \neq x$.

$e = e_1.m(e_2)$**:** by the typing rule for method invocation, we have that $\Gamma \vdash e_1 : [m : \hat{\alpha} \to \alpha]$ and $\Gamma \vdash e_2 : \hat{\alpha}$. By inductive hypothesis, $\Gamma' \vdash e_1[^{e'}/_x] : [m : \hat{\alpha} \to \alpha]$ and $\Gamma' \vdash e_2[^{e'}/_x] : \hat{\alpha}$; hence $\Gamma' \vdash (e_1.m(e_2))[^{e'}/_x] : \alpha$, since $m \neq x$.

$e =$ **new** $C(\widetilde{e})$**:** by the typing rule for object creation, we have that $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\Gamma \vdash \widetilde{e} : \widetilde{\alpha'}$, for $\widetilde{\alpha'} \leq \widetilde{\alpha}$, and $\alpha = [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[\widetilde{l_i : \tau_i}] \neq \mathbf{0}$. By inductive hypothesis, $\Gamma' \vdash \widetilde{e}[^{e'}/_x] : \widetilde{\alpha'}$; hence $\Gamma' \vdash$ (**new** $C(\widetilde{e})$)$[^{e'}/_x] : \alpha$, since $C \neq x$.

□

Theorem 2 (Subject reduction). *If* $\vdash e : \alpha$ *and* $e \to e'$*, then* $\vdash e' : \alpha$.

Proof. The proof is by induction on the derivation for $e \to e'$. There are the following base cases.

- $e = (\textbf{new } C(\widetilde{u})).a$ and $e' = u_i$, where $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ and $a = a_i$. By the typing rule for field access, we have that $\vdash \textbf{new } C(\widetilde{u}) : [a : \alpha]$. By the typing rule for **new**, $\vdash \textbf{new } C(\widetilde{u}) : [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \wedge \bigwedge_j \neg[\widetilde{l_j : \tau_j}]$, for $\widetilde{\alpha'} \leq \widetilde{\alpha}$. By subsumption, the two typing judgements for **new** $C(\widetilde{u})$ are compatible only if $[\widetilde{a : \alpha'}, \widetilde{m : \mu}] \wedge \bigwedge_j \neg[\widetilde{l_j : \tau_j}] \leq [a : \alpha]$, i.e., $\alpha'_i \leq \alpha$. Thus, $\vdash e' : \alpha'_i$ and, again by subsumption, $\vdash e' : \alpha$, as desired.

- $e = (\textbf{new } C(\widetilde{u})).m(u)$ and $e' = e''[{}^{u}/_{x}, {}^{\textbf{new } C(\widetilde{u})}/_{\textbf{this}}]$, where $body(m, u, C) = \lambda x.e''$. Since e is typeable, it must be that $\vdash \textbf{new } C(\widetilde{u}) : [m : \hat{\alpha} \to \alpha]$ and $\vdash u : \hat{\alpha}$. The first typing judgement entails that $\vdash \textbf{new } C(\widetilde{u}) : \alpha'$, for $[m : \hat{\alpha} \to \alpha] \geq \alpha' = [\widetilde{a : \alpha''}, \widetilde{m : \mu}] \wedge \bigwedge_j \neg[\widetilde{l_j : \tau_j}] \not\simeq \mathbf{0}$; moreover, $type(C) = [\widetilde{a : \alpha'}, \widetilde{m : \mu}]$ and $\vdash \widetilde{u} : \widetilde{\alpha''}$, for $\widetilde{\alpha''} \leq \widetilde{\alpha'}$. Second, let $C = D_0$ **extends** D_1 **extends** $\ldots$ **extends** $D_n = Object$ be the path in the class hierarchy from C to $Object$, for some $n \geq 0$. Now, $body(m, u, C) = \lambda x.e''$ entails that there exists $i \in \{0, \ldots, n-1\}$ such that D_i contains the method definition $\alpha_i\ m(\hat{\alpha}_i\ x)\{\textbf{return } e''\}$, for $\vdash u : \hat{\alpha}_i$; moreover, D_i is the class closest to C in the hierarchy that satisfies this fact. By typeability, it holds that $\vdash_{D_i} \alpha_i\ m(\hat{\alpha}_i\ x)\{\textbf{return } e''\}$ and, hence, $x : \hat{\alpha}_i, \textbf{this} : type(D_i) \vdash e'' : \alpha_i$. Since $\alpha' \leq type(C)$, by Lemma 2 and by Lemma 4 applied to **this** and to x, we have that $\vdash e''[{}^{u}/_{x}, {}^{\textbf{new } C(\widetilde{u})}/_{\textbf{this}}] : \alpha_i$. To conclude, it suffices to show that $\alpha_i \leq \alpha$. This fact holds because $\alpha' \leq [m : \hat{\alpha} \to \alpha]$ and the μ for m contains the conjunct $(\hat{\alpha}_i \setminus \hat{\alpha}_{i-1}) \setminus \cdots \setminus \hat{\alpha}_0 \to \alpha_i$, where $\hat{\alpha}_j \to \alpha_j$ is the type declared for m in D_j.

For the inductive step, we reason by case analysis on the last rule used in the inference. We have four possible cases.

- $e = e_1.a$, $e_1 \to e_2$ and $e' = e_2.a$. By the typing rule for field access, $\vdash e_1 : [a : \alpha]$, for some α, and, by induction, $\vdash e_2 : [a : \alpha]$. Again by the typing rule for field access, $\vdash e' : \alpha$.

- $e = e_1.m(u)$, $e_1 \to e_2$ and $e' = e_2.m(u)$. By the typing rule for method invocation, $\vdash e_1 : [m : \hat{\alpha} \to \alpha]$ and $\vdash u : \hat{\alpha}$. By induction, $\vdash e_2 : [m : \hat{\alpha} \to \alpha]$. Again by the typing rule for method invocation, $\vdash e' : \alpha$.

- $e = e_0.m(e_1)$, $e_1 \to e_2$ and $e' = e_0.m(e_2)$. By the typing rule for method invocation, $\vdash e_0 : [m : \hat{\alpha} \to \alpha]$ and $\vdash e_1 : \hat{\alpha}$. By induction, $\vdash e_2 : \hat{\alpha}$. Again by the typing rule for method invocation, $\vdash e' : \alpha$.

- $e = \textbf{new } C(e_1, \ldots, e_i, \ldots, e_k)$, $e_i \to e'_i$ and $e' = \textbf{new } C(e_1, \ldots, e'_i, \ldots, e_k)$. By the typing rule for **new**, $\vdash e_i : \alpha_i$. By induction, $\vdash e'_i : \alpha_i$. Again by the typing rule for **new**, $\vdash e' : \alpha$. □

Theorem 3 (Progress). *If* $\vdash e : \alpha$ *and* e *is a closed expression, then* e *is a value or there exists* e' *such that* $e \to e'$.

Proof. The proof is by induction on the structure of e. Since e is closed, the only possible base case is when e is a basic value, and in this case the claim trivially holds. For the inductive step, we reason by case analysis on the form of e.

- $e = e_0.a$. By the typing rule for field access, $\vdash e_0 : [a : \alpha]$; by induction, either e_0 is a value (and in this case it must be $e_0 = \textbf{new}\ C(\widetilde{u})$, for some C and $\widetilde{u}$) or $e_0 \to e_0'$, for some e_0'. In the second case, we easily conclude that $e \to e'$, by letting $e' = e_0'.a$. In the first case, by the typing rule for **new**, $\vdash \textbf{new}\ C(\widetilde{u}) : [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \wedge \bigwedge_j \neg[\widetilde{l_j : \tau_j}]$, where $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\vdash \widetilde{u} : \widetilde{\alpha'}$ and $\widetilde{\alpha'} \leq \widetilde{\alpha}$. Moreover, $[\widetilde{a : \alpha'}, \widetilde{m : \mu}] \wedge \bigwedge_j \neg[\widetilde{l_j : \tau_j}] \leq [a : \alpha]$ implies that there exists i such that $a = a_i$; thus, $e \to e'$, where $e' = u_i$.

- $e = e_0.m(e_1)$. By the typing rule for method invocation, $\vdash e_0 : [m : \hat{\alpha} \to \alpha]$. Like in the previous case, the interesting part is when e_0 is a value (in particular, $e_0 = \textbf{new}\ C(\widetilde{u})$, for some C and $\widetilde{u}$); with a similar reasoning, we can say that $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, that $\vdash \widetilde{u} : \widetilde{\alpha'}$ for $\widetilde{\alpha'} \leq \widetilde{\alpha}$, that $\vdash e_1 : \hat{\alpha}$ and $m = m_i$, for some i.

  Let us first consider the case when e_1 is not a value. By induction, there exists a e_1' such that $e_1 \to e_1'$; hence, $e \to e'$, by letting $e' = e_0.m(e_1')$.

  So, let us now assume that $e_1 = u$. By definition of function *type*, it must be that $\mu_i = (\hat{\alpha}^0 \to \alpha^0) \wedge \bigwedge_{i=1}^n (\hat{\alpha}^i \setminus \hat{\alpha}^0 \to \alpha^i) \leq \bigwedge_{i=1}^n (\hat{\alpha}^i \to \alpha^i) = \alpha'(m)$, where C extends D in $\widetilde{L}$ and $type(D) = \alpha'$. Also, by subtyping, $\mu_i \leq \hat{\alpha} \to \alpha$. It now suffices to prove that $body(m, u, C) = \lambda x.e''$, for some e'' and x. First, notice that C cannot be $Object$, otherwise $\vdash e : \alpha$ could not hold for any α. Then, we work by a second induction over the distance between D and $Object$ in the class hierarchy defined by $\widetilde{L}$. The base case is when D is $Object$: then, $\alpha_i'(m) = \hat{\alpha}^0 \to \alpha^0$ and, hence, $\hat{\alpha} \leq \hat{\alpha}^0$. By subsumption, $\vdash u : \hat{\alpha}^0$ and hence $body(m, u, C) = \lambda x.e''$, where C contains the method declaration $\alpha^0\ m(\hat{\alpha}^0\ x)\{\textbf{return}\ e''\}$. For the inductive step, if $\alpha'(m) = \mathbf{0} \to \mathbf{1}$, we work like in the case when $D = Object$; otherwise, by (second) induction, we know that $body(m, u, D) = \lambda x.e''$. If C does not contain any declaration for method m, this suffices to conclude; otherwise, let $\alpha^0\ m(\hat{\alpha}^0\ x)\{\textbf{return}\ \ e'''\}$ be such a declaration. If $\vdash u : \hat{\alpha}^0$, then $body(m, u, D) = \lambda x.e'''$; otherwise, $body(m, u, D) = \lambda x.e''$. This suffices to conclude.

- $e = \textbf{new}\ C(\widetilde{e})$. If $\widetilde{e}$ is a sequence of values, then e is a value. Otherwise, there is an i such that e_i is not a value; by induction, we have that $e_i \to e_i'$ and hence we conclude by letting $e' = \textbf{new}\ C(e_1, \ldots, e_i', \ldots, e_k)$. □

8 Discussion on the calculus

8.1 Recursive class definitions

It is possible to write recursive class definitions by assuming a special basic value **null** and a corresponding basic type **unit**, having **null** as its only value. In Java, it is assumed that **null** belongs to every class type; here, because of the complex types we are working with (mainly, because of negations), this assumption is not valid. This, however, enables us to specify when a field can/cannot be **null**; this is similar to what

happens in database systems. In particular, lists of integers can now be defined as:

$$
\begin{array}{ll}
L_{intList} = & \textbf{class } intList \textbf{ extends } Object \{ \\
& \quad \textbf{int } val; \\
& \quad (\alpha \vee \textbf{unit}) \; succ; \\
& \quad intList \; (\textbf{int } x, (\alpha \vee \textbf{unit}) \; y)\{\textbf{this}.val = x; \textbf{this}.succ = y\} \\
& \quad \ldots \\
& \}
\end{array}
$$

where α is the regular tree representing the solution of the recursive type equation

$$\alpha = [val : \textbf{int}, succ : (\alpha \vee \textbf{unit})]$$

Now, we can create the list $\langle 1, 2 \rangle$ as the value **new** *intList*$(1, \textbf{new } intList(2, \textbf{null}))$.

8.2 Implementing Standard Multimethods

Usually in object oriented languages, multimethods can be defined within a single class. For simplicity, we have defined a language where at most one definition can be given for a method name in a class. It is however possible to partially encode multimethods by adding one auxiliary subclass for every method definition. For instance, suppose we want to define a multimethod *m* twice within class *A*:

$$
\begin{array}{l}
\textbf{class } A \textbf{ extends } B \{ \\
\quad \ldots \\
\quad \alpha_1 \; m \; (\alpha'_1 \; x)\{\textbf{return } e_1\} \\
\quad \alpha_2 \; m \; (\alpha'_2 \; x)\{\textbf{return } e_2\} \\
\}
\end{array}
$$

We then replace it with the following declarations:

$$
\begin{array}{ll}
\textbf{class } A1 \textbf{ extends } B \{ & \textbf{class } A \textbf{ extends } A1 \{ \\
\quad \ldots & \quad \alpha_2 \; m \; (\alpha'_2 \; x)\{\textbf{return } e_2\} \\
\quad \alpha_1 \; m \; (\alpha'_1 \; x)\{\textbf{return } e_1\} & \} \\
\}
\end{array}
$$

Introducing subclasses is something that must be done with care. Indeed, it is not guaranteed, in general, that the restrictions for the definition of function *type* (see Definition 10) are always satisfied. So, in principle, the encoding described above could turn a class hierarchy where the function *type* is well-defined into a hierarchy where it is not. However, this situation never arises if different bodies of a multimethod are defined for inputs of mutually disjoint types, as we normally do. Also, it is not difficult to add to the language a *typecase* construct, similar to the one of ℂDuce, that would allow more expressivity. We did not pursue this approach in the present paper to simplify the presentation.

8.3 Implementing Typical Java-like Constructs

We now briefly show how we can implement in our framework traditional programming constructs like *if-then-else*, (a structural form of) *instanceof* and *exceptions*. Other constructs, like sequential composition and loops, can also be defined.

The expression **if** e **then** e_1 **else** e_2 can be implemented by adding to the program the class definition (using the standard multimethods described in §8.2):

$$\begin{array}{l}\textbf{class } \mathit{Test} \textbf{ extends } \mathit{Object} \{\\ \quad \alpha\ m\ (\{\textbf{true}\}\ x)\{\textbf{return } e_1\}\\ \quad \alpha\ m\ (\{\textbf{false}\}\ x)\{\textbf{return } e_2\}\\ \}\end{array}$$

where {**true**} and {**false**} are the singleton types containing only values **true** and **false**, respectively, and α is the type of e_1 and e_2. Then, **if** e **then** e_1 **else** e_2 can be simulated by (**new** *Test*()).$m(e)$. Notice that this term typechecks, since *test* has type $[m : (\{\textbf{true}\} \to \alpha) \wedge (\{\textbf{false}\} \to \alpha)] \simeq [m : (\{\textbf{true}\} \vee \{\textbf{false}\}) \to \alpha] \simeq [m : \textbf{bool} \to \alpha]$. Indeed, in [7] it is proved that $(\alpha_1 \to \alpha) \wedge (\alpha_2 \to \alpha) \simeq (\alpha_1 \vee \alpha_2) \to \alpha$ and, trivially, $\{\textbf{true}\} \vee \{\textbf{false}\} \simeq \textbf{bool}$.

The construct e **instanceof** α checks whether e is typeable at α and can be implemented in a way similar to the *if-then-else*:

$$\begin{array}{l}\textbf{class } \mathit{InstOf} \textbf{ extends } \mathit{Object} \{\\ \quad \textbf{bool } m_{\alpha_1}(\alpha_1\ x)\{\textbf{return true}\}\\ \quad \textbf{bool } m_{\alpha_1}(\neg\alpha_1\ x)\{\textbf{return false}\}\\ \quad \ldots\\ \quad \textbf{bool } m_{\alpha_k}(\alpha_k\ x)\{\textbf{return true}\}\\ \quad \textbf{bool } m_{\alpha_k}(\neg\alpha_k\ x)\{\textbf{return false}\}\\ \}\end{array}$$

where $\alpha_1, \ldots, \alpha_k$ are the types occurring as arguments of an **instanceof** in the program. Then, e **instanceof** α can be simulated by (**new** *InstOf*()).$m_\alpha(e)$.

Finally, **try** e **catch**($\alpha\ x$) e' evaluates e and, if an exception of type α is raised during the evaluation, expression e' is evaluated. First of all, we assume that every exception is an object of a subclass of class *Exception* that, in turn, extends *Object*. Second, every method that can raise an exception of type α must specify this fact in the return type (this resembles the use of the **throws** keyword in Java); in particular, if m's type is $\alpha_1 \to \alpha_2$ and it can raise exceptions of type α, it should be declared as

$$(\alpha \vee \alpha_2)\ \ m(\alpha_1\ x)\{\ldots\}$$

Indeed, every statement **throw** e within m will be translated in our framework as **return** e. Third, we can translate **try** e **catch**($\alpha\ x$) e' as

$$\textbf{let } x = e \textbf{ in } (\textbf{if } (x \textbf{ instanceof } \alpha) \textbf{ then } e' \textbf{ else } x)$$

Here, we assume a standard construct **let** $y = e_1$ **in** e_2; it can be implemented in our framework as

$$\textbf{this}.\mathit{let}(e_1)$$

once we have added to the class the method

$$\alpha_2\ \mathit{let}(\alpha_1\ y)\{\textbf{return } e_2\}$$

where α_1 and α_2 are the types of e_1 and e_2, respectively.

8.4 Nominal subtyping vs. Structural subtyping

Semantic subtyping is a powerful typing discipline, but explicitly annotating programs with structural types could be too cumbersome for programmers Thus, we can introduce aliases. We could write

$$
\begin{array}{lll}
L'_{intList} = & \textbf{class } intList \textbf{ extends } Object \{ & \\
& \quad \textbf{int } val; & \\
& \quad (intList \vee \textbf{unit}) \; succ; & \\
& \quad intList \; (\textbf{int } x, (intList \vee \textbf{unit}) \; y)\{\textbf{this}.val = x; \textbf{this}.succ = y\} & \\
& \quad \ldots & \\
& \} &
\end{array}
$$

instead of $L_{intList}$ in § 8.1. Any sequence of class declarations written in this extended syntax can be then compiled into the standard syntax in two steps:

- First, extract from the sequence of class declarations a system of (mutually recursive) type declarations; in doing this, every class name should be considered as a type identifier. Then, solve such a system of equations.
- Second, replace every occurrence of every class name occurring in a type position (i.e., not in a class header nor as the name of a constructor) with the corresponding solution of the system.

For example, the system of equations (actually, made up of only one equation) associated with $L'_{intList}$ is $intList = [val : \textbf{int}, succ : (intList \vee \textbf{unit})]$; if we assume that α denotes the solution of such an equation, the class declaration resulting at the end of the compilation is exactly $L_{intList}$ in § 8.1.

But nominal types can be more powerful than just shorthands. When using structural subtyping, we can interchangeably use two different classes having the very same structure but different names. However, there can be programming scenarios where also the name of the class (and not only its structure) could be needed. A typical example is the use of exceptions, where one usually extends class *Exception* without changing its structure. In such cases, nominal subtyping can be used to enforce a stricter discipline. Another sensible use of nominal types is for expressing design intent. Here, the classical example is

$$
\textbf{class } Shape \; \{\textbf{boolean } draw()\} \qquad\qquad \textbf{class } Cowboy \; \{\textbf{boolean } draw()\}
$$

where the two classes have different semantics even though equal structural types, and freely mixing *Cowboy* and *Shape* objects would not be semantically correct.

We can integrate this form of nominal subtyping in our semantic framework. To do that, we add to each class a hidden field that represents all the nominal hierarchy that can be generated by that class. If we want to be nominal, we will consider also this hidden field while checking subtyping. In practice, the (semantic) 'nominal' type of a class is the set of qualified names of all its subclasses; this will enable us to say that C is a 'nominal' subtype of D if and only if C's subclasses form a subset of D's ones. Notice that working with subsets is the key feature of our semantic approach to subtyping. This is the reason why we need types as sets and, e.g., cannot simply add to objects a field with the class they are instance of.

Let us denote with CN the (countable) set of class names. An element of CN^* can be thought of as a partially qualified name of a class – fully qualified if it starts with *Object*. We consider now sets of qualified names, ranged over by X, Y, Z. They will

be used as types, the subtyping being defined as set inclusion. For each class C, we consider the type

$$X_C = \left\{ \begin{array}{l} \\ s_1 s_2 \in CN^* : \\ \\ \end{array} \begin{array}{l} s_1 = C_0 \ldots C_k \;\&\; k \geq 0 \;\&\; C_0 = \mathit{Object} \;\&\; C_k = C \\ \&\; \forall\, i \in \{0, \ldots, k-1\}. C_{i+1} \textbf{ extends } C_i \\ \&\; s_2 \in (CN \setminus \{C_0, \ldots, C_k\})^* \end{array} \right\}$$

Following the above intuition, X_C contains the fully qualified class names of all the potential subclasses of C. Finally, we choose a special reserved name **name** that cannot occur in the program. This will be the name of the "hidden" nominal field. For example, take a standard example of Java inheritance, where class *Object* is extended by class *Point* that is in turn extended by classes *ColPoint*, of coloured points, and *GeomPoint*, of geometrical points. We can say, e.g., that the third class is a (nominal) subtype of the second one by noting that:

$$\begin{array}{rcl} X_{\mathit{Point}} & = & \{\mathit{Object.Point}, \\ & & \quad \mathit{Object.Point.ColPoint}, \\ & & \quad\quad \mathit{Object.Point.ColPoint.Pixel}, \\ & & \quad\quad\quad \ldots \\ & & \quad\quad \mathit{Object.Point.ColPoint.3DPoint}, \\ & & \quad\quad\quad \ldots \\ & & \quad\quad \ldots, \\ & & \quad \mathit{Object.Point.GeomPoint}, \\ & & \quad\quad \mathit{Object.Point.GeomPoint.Circle}, \\ & & \quad\quad\quad \ldots \\ & & \quad\quad \mathit{Object.Point.GeomPoint.Line}, \\ & & \quad\quad\quad \ldots \\ & & \quad\quad \ldots, \\ & & \quad \ldots \\ & & \} \end{array}$$

$$\begin{array}{rcl} X_{\mathit{ColPoint}} & = & \{\mathit{Object.Point.ColPoint}, \\ & & \quad \mathit{Object.Point.ColPoint.Pixel}, \\ & & \quad\quad \ldots \\ & & \quad \mathit{Object.Point.ColPoint.3DPoint}, \\ & & \quad\quad \ldots \\ & & \quad \ldots \\ & & \} \end{array}$$

Indeed, $X_{\mathit{ColPoint}} \subseteq X_{\mathit{Point}}$.

Now, given a sequence of class declarations $\widetilde{L}$, we denote with $(\widetilde{L})^{\mathbf{name}}$ the sequence obtained by adding to every class declaration for class C in $\widetilde{L}$ the field declaration

$$X_C \;\; \textbf{name}$$

It is easy to verify the following desirable facts

- C is a sub-class of D if and only if $\mathit{type}_{(\widetilde{L})^{\mathbf{name}}}(C) \leq \mathit{type}_{(\widetilde{L})^{\mathbf{name}}}(D)$;
- For every C, it holds that $\mathit{type}_{(\widetilde{L})^{\mathbf{name}}}(C) \leq \mathit{type}^{-}_{\widetilde{L}}(C)$;
- If $\mathit{type}^{-}_{\widetilde{L}}(C) \simeq \mathit{type}^{-}_{\widetilde{L}}(D)$ but $C \neq D$, then $\mathit{type}_{(\widetilde{L})^{\mathbf{name}}}(C) \neq \mathit{type}_{(\widetilde{L})^{\mathbf{name}}}(D)$.

where the subscript to the function *type* specifies the declarations in which the function is calculated.

By the way, notice that here we are working with infinite sets. But these sets have always a finite representation that makes the subtyping still decidable. Indeed, every set X_C can be represented by the fully qualified name of C and C is a sub-class of D if and only if X_D is a prefix of X_C.

It remains to describe how we can use nominal subtyping in place of the structural one. We propose two ways. In declaring a class or a field, or in the return type of a method, we could add the keyword **nominal**, to indicate to the compiler that nominal subtyping should always be used with it. However, the only place where subtyping is used is in function *body*, i.e. when deciding which body of an overloaded method we have to activate on a given sequence of actual values. Therefore, we could be even more flexible, and use the keyword **nominal** in method declarations, to specify which method arguments have to be checked nominally and which ones structurally. For example, consider the following class declaration:

$$\begin{array}{l}\textbf{class } A \textbf{ extends } \mathit{Object} \{ \quad \ldots \\ \quad \textbf{int } m\ (C\ x,\ \textbf{nominal } C\ y)\{\ \textbf{return } 0;\ \} \\ \}\end{array}$$

Here, every invocation of method m will check the type of the first argument structurally and the type of the second one nominally. This is a mechanism akin to the notion of *brand* in Strongtalk [36]. Thus, if we consider the following class declarations

$$\textbf{class } C \textbf{ extends } \mathit{Object} \{\ \} \qquad\qquad \textbf{class } D \textbf{ extends } \mathit{Object} \{\ \}$$

the expressions

$$(\textbf{new } A()).m(\textbf{new } C(), \textbf{new } C())$$

$$(\textbf{new } A()).m(\textbf{new } D(), \textbf{new } C())$$

$$(\textbf{new } A()).m(\textbf{new } \mathit{Object}(), \textbf{new } C())$$

typecheck, whereas

$$(\textbf{new } A()).m(\textbf{new } C(), \textbf{new } D()) \qquad (\textbf{new } A()).m(\textbf{new } C(), \textbf{new } \mathit{Object}())$$

do not.

In practice, for each sequence of class declarations $\widetilde{L}$, the compiler will build the types both for $\widetilde{L}$ and for $(\widetilde{L})^{\textbf{name}}$, and will decide which one to use according the presence or not of the keyword **nominal**.

9 Conclusions and Future Work

We have presented a Java-like programming framework that integrates structural subtyping, boolean connectives and semantic subtyping to exploit and combine the benefits of such approaches. There is still work to do in this research line.

This paper lays out the foundations for a concrete implementation of our framework. First of all, a concrete implementation calls for algorithms to decide the subtyping relation; then, decidability of subtyping is exploited to define a typechecking algorithm for our type system. This can be achieved by adding algorithms similar to those in [7]. These are intermediate steps towards a prototype programming environment where writing and evaluating the performances of code written in the new formalism. Of course, extending a real language as Java with structural types and multi-methods is far from being a simple task: this would imply major modifications of the Java Virtual

Machine (which is quite a complex architecture) and would pose non trivial efficiency and legacy code issues. Our present paper lays the foundations of the typing mechanisms, but deliberately neglects many other important problems that would arise in practice when trying to implement the proposed extension.

Another direction for future research is the enhancement of the language considered. For example, one can consider the extension of FJ with assignments; this is an important aspect because mutable values are crucial for modeling the heap, a key feature in object oriented programming. We think that having a state would complicate the issue of typing, because of the difference between the declared and the actual type of an object. Some ideas on how to implement the mutable state can come from the choices made in the implementation of $\mathbb{C}$Duce.

Another possibility for enhancing the language is the introduction of higher-order values, in the same vein as the Scala programming language [37]; since the framework of [7] is designed for a higher-order language, the theoretical machinery developed therein should be adaptable to the new formalism.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proc. of FPCA*, pages 31–41. ACM, 1993.

[2] F. M. Damm. Subtyping with union types, intersection types and recursive types. In *Proc. of TACS*, pages 687–706. Springer, 1994.

[3] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *SIGPLAN Notices*, 36(3):67–80, 2001.

[4] H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[5] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proc. ICFP*, pages 51–63, 2003.

[6] G. Castagna. Semantic subtyping: Challenges, perspectives, and open problems. In *Proc. of ICTCS*, pages 1–20, 2005.

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

[8] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet and L. Padovani. Polymorphic Functions with Set-Theoretic Types. Part 1: Syntax, Semantics, and Evaluation. In *Proc. of POPL*, pages 5–17, 2014.

[9] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2003.

[10] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the pi-calculus. *Theoretical Computer Science*, 398(1-3):217–242, 2008.

[11] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15:575–631, 1993.

[12] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. In *Proc. POPL*, pages 419–428. ACM, 1993.

[13] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proc. FPCA*, pages 31–41, 1993.

[14] D Ancona and G Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into horn formulas. In *Proc. GANDALF*, pages 214–230, 2010.

[15] D. Ancona and A. Corradi. Sound and complete subtyping between coinductive types for object-oriented languages. In *Proc. ECOOP*, pages 282–307. Springer, 2014.

[16] M. Bonsangue, J. Rot, D. Ancona, F. de Boer, and J. Rutten. A coalgebraic foundation for coinductive union types. In *Proc. of ICALP*, pages 62–73, 2014.

[17] V. Gapeyev, M. Y. Levin, and B. Pierce. Recursive subtyping revealed. *JFP*, 12(6):511–548, 2002.

[18] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proc. TLCA*, pages 63–81, 1997.

[19] D. Kozen and A. Silva. Practical coinduction. *Mathematical Structures in Computer Science*, pages 1–21, 2016.

[20] D. Pearce. Sound and Complete Flow Typing with Unions, Intersections and Negations, In *Proc. of VMCAI*, pages 335–354, 2013.

[21] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[22] O. Dardha, D. Gorla, and D. Varacca. Semantic Subtyping for Objects and Classes. In *Proc. of FMOODS/FORTE*, pages 66–82. Springer, 2013.

[23] J.T. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In *Proc. of ECOOP*, volume 1098 of *LNCS*, pages 3–25. Springer, 1996.

[24] G. Castagna. *Object-oriented programming: a unified foundation*. Progress in Theoretical Computer Science series, Birkäuser, Boston 1997.

[25] J.T. Boyland and G. Castagna. Parasitic Methods: an implementation of multi-methods for Java. In *Proc. of OOPSLA*. ACM Press, 1997.

[26] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *Proc. of ECOOP*, pages 364–388. Springer, 2004.

[27] J. Gil and I. Maman. Whiteoak: introducing structural typing into Java. In *Proc. of OOPSLA*, pages 73–90. ACM, 2008.

[28] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.11*, 2008.

[29] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proc. of ECOOP*, pages 260–284. Springer, 2008.

[30] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In *Proc. of ESOP*, pages 95–111. Springer, 2009.

[31] K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.

[32] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. of POPL*, pages 40–53. ACM, 1997.

[33] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *Proc. of ECOOP*, pages 2–26. Springer, 2009.

[34] M. Abadi and L. Cardelli. A Theory of Primitive Objects - Untyped and First-Order Systems. In *Proc. of TACS*, pages 296–320. Springer, 1994.

[35] R. Agrawal, L.G. de Michiel and B. G. Lindsay. Static type checking of multi-methods. In *Proc. of OOPSLA*, pages 113–128. ACM Press, 1991.

[36] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

[37] M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, 2004.

Appendix: Technical Proofs for Closing the Circle

The appendix is fully devoted to prove the main theoretical result of this paper, namely Theorem 1. To this aim, we first give the definition of *disjunctive normal form* (DNF), which gives us a systematic and uniform way of writing types; this will then facilitate the presentation of our theoretical development. Then, we show that $[\![\cdot]\!]_{\mathcal{V}}$ is a set-theoretic interpretation; this fact will allow us to use standard set-theoretic equalities when working with such an interpretation. Finally, using these notions and results, we shall prove that $\leq_{\mathcal{V}} = \leq_{\mathcal{B}}$.

A Disjunctive Normal Forms for Types

A disjunctive normal form is based on the notion of atomic types, which are the most basic form of types. Recall from §3.3 that there are three kinds of atomic types $\mathbb{T}_{\mathbf{basic}}$, $\mathbb{T}_{\mathbf{rec}}$ and $\mathbb{T}_{\mathbf{fun}}$ such that $\mathbb{T} = \mathbb{T}_{\mathbf{basic}} \uplus \mathbb{T}_{\mathbf{rec}} \uplus \mathbb{T}_{\mathbf{fun}}$. In order for a disjunctive normal form to be useful, we rely on the fact that every type can be decomposed as a disjunctive normal form and, conversely, that every disjunctive normal form corresponds to one precise type.

Definition 11. *A* disjunctive normal form δ *is a finite set of pairs of finite sets of atoms, namely an element of* $\mathcal{P}_f\left(\mathcal{P}_f(\mathbb{T}) \times \mathcal{P}_f(\mathbb{T})\right)$.

Next, we give the definition of the set-theoretic interpretation of a disjunctive normal form δ.

Definition 12. *If* $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(D)$ *is an arbitrary set-theoretic interpretation and* δ *a disjunctive normal form, we define* $[\![\delta]\!]$ *as:*

$$[\![\delta]\!] = \bigcup_{(P,N)\in\delta} \left(\bigcap_{t\in P} [\![t]\!] \cap \bigcap_{t\in N} (D \setminus [\![t]\!]) \right)$$

The set-theoretic interpretation separates the "positive" (P) and the "negative" (N) parts of the disjunctive normal form δ. By using standard set-theoretic equalities, we obtain:

$$\bigcup_{(P,N)\in\delta} \left(\bigcap_{t\in P} [\![t]\!] \cap \bigcap_{t\in N} (D \setminus [\![t]\!]) \right) = \bigcup_{(P,N)\in\delta} \left(\bigcap_{t\in P} [\![t]\!] \cap \bigcap_{t\in N} \overline{[\![t]\!]} \right) =$$
$$= \bigcup_{(P,N)\in\delta} \left(\bigcap_{t\in P} [\![t]\!] \cap \overline{\bigcup_{t\in N} [\![t]\!]} \right) = \bigcup_{(P,N)\in\delta} \left(\bigcap_{t\in P} [\![t]\!] \setminus \bigcup_{t\in N} [\![t]\!] \right)$$

In what follows, we shall usually use the latter expression for $[\![\delta]\!]$. Since an empty intersection is D and an empty union is $\emptyset$, we have that $[\![\delta]\!] \subseteq D$.

By following [7], we can prove, by construction, that for each $\tau \in \mathcal{T}$, it is possible to compute a disjunctive normal form $\mathcal{N}(\tau)$ such that $[\![\tau]\!] = [\![\mathcal{N}(\tau)]\!]$, for every set-theoretic interpretation $[\![\cdot]\!]$. In order to construct the disjunctive normal form $\mathcal{N}(\tau)$, we

associate with function $\mathcal{N}$ a second function $\mathcal{N}'$, both mutually defined as follows:

$$\begin{aligned}
\mathcal{N}(\mathbf{0}) &= \varnothing \\
\mathcal{N}(t) &= \{(\{t\}, \varnothing)\} \\
\mathcal{N}(\tau_1 \wedge \tau_2) &= \mathcal{N}(\tau_1) \cap \mathcal{N}(\tau_2) \\
\mathcal{N}(\neg\tau) &= \mathcal{N}'(\tau)
\end{aligned}$$

$$\begin{aligned}
\mathcal{N}'(\mathbf{0}) &= \{(\varnothing, \varnothing)\} \\
\mathcal{N}'(t) &= \{(\varnothing, \{t\})\} \\
\mathcal{N}'(\tau_1 \wedge \tau_2) &= \{(P_1 \cap P_2, N_1 \cap N_2) \mid (P_1, N_1) \in \mathcal{N}'(\tau_1), (P_2, N_2) \in \mathcal{N}'(\tau_2)\} \\
\mathcal{N}'(\neg\tau) &= \mathcal{N}(\tau)
\end{aligned}$$

It can be checked by induction over τ that $[\![\tau]\!] = [\![\mathcal{N}(\tau)]\!] = D \setminus [\![\mathcal{N}'(\tau)]\!]$.

Note that the inverse is also true: for every disjunctive normal form δ, there is a type τ such that $[\![\tau]\!] = [\![\delta]\!]$, for every set-theoretic interpretation $[\![\cdot]\!]$: it suffices to consider $\bigvee_{(P,N)\in\delta} (\bigwedge_{t\in P} t \setminus \bigvee_{t\in N} t)$. Hence, from now on we will interchangeably use the notions of type and disjunctive normal form.

B Types as Sets of Pseudo-Values

In this section we focus on the interpretation of types as sets of pseudo-values and we will show that this interpretation is set-theoretic. We state that a (pseudo-)value v and an atomic type t are *compatible* if they are of the same kind.

We will use the bootstrap model constructed in § 3.3, namely $[\![\cdot]\!]_{\mathcal{B}} : \mathcal{T} \to \mathcal{P}(\mathcal{B})$, and we write $\leq$ to indicate the subtyping relation induced by this model. Similarly, we write $\simeq$ to denote the corresponding equivalence relation, namely $\tau_1 \simeq \tau_2 \iff [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$ and $[\![\tau_2]\!]_{\mathcal{B}} \subseteq [\![\tau_1]\!]_{\mathcal{B}}$.

Lemma 5. $[\![\mathbf{0}]\!]_{\mathcal{V}} = \varnothing$.

Proof. It suffices to prove that $\vdash v : \tau \Longrightarrow \tau \not\simeq \mathbf{0}$, for every pseudo-value v. We reason by case analysis on v. The cases to consider are when v is a constant, an object creation or an abstraction; in all cases the result trivially follows. □

Lemma 6. *If $\tau_1 \leq \tau_2$ then $[\![\tau_1]\!]_{\mathcal{V}} \subseteq [\![\tau_2]\!]_{\mathcal{V}}$. Thus, if $\tau_1 \simeq \tau_2$ then $[\![\tau_1]\!]_{\mathcal{V}} = [\![\tau_2]\!]_{\mathcal{V}}$.*

Proof. Follows immediately by the subsumption rule. □

Lemma 7. $[\![\tau_1 \wedge \tau_2]\!]_{\mathcal{V}} = [\![\tau_1]\!]_{\mathcal{V}} \cap [\![\tau_2]\!]_{\mathcal{V}}$.

Proof. By Lemma 6, we have that $[\![\tau_1 \wedge \tau_2]\!]_{\mathcal{V}} \subseteq [\![\tau_i]\!]_{\mathcal{V}}$, for $i \in \{1, 2\}$; this imples that $[\![\tau_1 \wedge \tau_2]\!]_{\mathcal{V}} \subseteq [\![\tau_1]\!]_{\mathcal{V}} \cap [\![\tau_2]\!]_{\mathcal{V}}$.

For the opposite inclusion, first observe that, if $[\![\tau_1]\!]_{\mathcal{V}} \cap [\![\tau_2]\!]_{\mathcal{V}} = \varnothing$, then the inclusion trivially holds. So, let us assume that there exists $v \in [\![\tau_1]\!]_{\mathcal{V}} \cap [\![\tau_2]\!]_{\mathcal{V}}$, i.e., $\vdash v : \tau_1$ and $\vdash v : \tau_2$. Without loss of generality, we can assume that such type derivations both end with an instance of *(const)/(new)/(abstr)* followed by an instance of *(subsum)* (such an instance may also be useless, in the sense that it assigns the type in the premise also to the conclusion). We now reason on the possible kind of v:

$v = c$: In this case, the only possible typing axiom used to type c is *(const)* and it states that $\vdash c : \mathbb{B}_c$; hence, the only way to infer τ_1 and τ_2 for c is when $\mathbb{B}_c \leq \tau_1$ and $\mathbb{B}_c \leq \tau_2$, i.e., $\mathbb{B}_c \leq \tau_1 \wedge \tau_2$. Hence, $\vdash v : \tau_1 \wedge \tau_2$, as desired.

$v = \mathbf{new}\ C(\widetilde{u})$**:** Here, the typing rule used is *(new)*, with $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\Gamma \vdash \widetilde{u} : \widetilde{\alpha'}$, for $\widetilde{\alpha'} \leq \widetilde{\alpha}$, and

$$\alpha'_1 = [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \ \wedge \bigwedge_{i=1\ldots n} \neg[\widetilde{l_i : \tau_i}] \not\simeq \mathbf{0}$$

$$\alpha'_2 = [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \ \wedge \bigwedge_{i=n+1\ldots n'} \neg[\widetilde{l_i : \tau_i}] \not\simeq \mathbf{0}$$

Moreover, $\alpha'_1 \leq \tau_1$ and $\alpha'_2 \leq \tau_2$. Let us now define α' as follows:

$$\alpha' = [\widetilde{a : \alpha'}, \widetilde{m : \mu}] \ \wedge \bigwedge_{i=1\ldots n'} \neg[\widetilde{l_i : \tau_i}]$$

Trivially, $\alpha' \simeq \alpha'_1 \wedge \alpha'_2 \not\simeq \mathbf{0}$ and $\alpha'_1 \wedge \alpha'_2 \leq \tau_1 \wedge \tau_2$. We can now use rules *(new)* and *(subsum)* to deduce $\vdash \mathbf{new}\ C(\widetilde{u}) : \tau_1 \wedge \tau_2$.

$v = \lambda_{(m,\rho)} x.e$**:** Similarly, the typing rule used is *(abstr)*, with $\rho(m) = \bigwedge_{i\in I}(\alpha_i \to \alpha'_i)$, $x : \alpha_i \vdash e : \alpha'_i$, for all $i \in I$, and

$$\mu_1 = \bigwedge_{i\in\{1,\ldots,n\}} (\alpha_i \to \alpha'_i) \ \wedge \bigwedge_{j\in\{n+1,\ldots,m\}} \neg(\alpha_j \to \alpha'_j) \not\simeq \mathbf{0}$$

$$\mu_2 = \bigwedge_{i\in\{1,\ldots,n\}} (\alpha_i \to \alpha'_i) \ \wedge \bigwedge_{j\in\{m+1,\ldots,h\}} \neg(\alpha_j \to \alpha'_j) \not\simeq \mathbf{0}$$

Moreover, $\mu_1 \leq \tau_1$ and $\mu_2 \leq \tau_2$. Let us now define μ as follows:

$$\mu = \bigwedge_{i\in\{1,\ldots,n\}} (\alpha_i \to \alpha'_i) \ \wedge \bigwedge_{j\in\{n+1,\ldots,h\}} \neg(\alpha_j \to \alpha'_j)$$

Trivially, $\mu \simeq \mu_1 \wedge \mu_2 \not\simeq \mathbf{0}$ and $\mu_1 \wedge \mu_2 \leq \tau_1 \wedge \tau_2$; by rule *(abstr)* and *(subsum)*, $\vdash \lambda_{(m,\rho)} x.e : \tau_1 \wedge \tau_2$. □

Lemma 8. $[\![\neg\tau]\!]_{\mathcal{V}} = \mathcal{V} \setminus [\![\tau]\!]_{\mathcal{V}}$.

Proof. Trivially, we have $\tau \wedge \neg\tau \simeq \mathbf{0}$; by Lemma 7, 6 and 5, $[\![\tau]\!]_{\mathcal{V}} \cap [\![\neg\tau]\!]_{\mathcal{V}} = [\![\tau \wedge \neg\tau]\!]_{\mathcal{V}} = [\![\mathbf{0}]\!]_{\mathcal{V}} = \emptyset$. It then remains to prove that $[\![\tau]\!]_{\mathcal{V}} \cup [\![\neg\tau]\!]_{\mathcal{V}} = \mathcal{V}$, i.e.

$$\forall v \in \mathcal{V}.\ (\vdash v : \tau) \text{ or } (\vdash v : \neg\tau)$$

To this aim, let us define $T_v = \{\tau \in \mathcal{T} \mid \vdash v : \tau \text{ or } \vdash v : \neg\tau\}$. If we now prove that $T_v = \mathcal{T}$, for every $v \in \mathcal{V}$, we have done. Indeed, for every $v \in \mathcal{V}$, it holds that $\tau \in \mathcal{T} = T_v$ and this may hold either because $\vdash v : \tau$ or because $\vdash v : \neg\tau$.

Trivially, T_v is closed under $\neg$. Moreover, by *(subsum)*, $\vdash v : \mathbf{1} = \neg\mathbf{0}$ and hence $\mathbf{0} \in T_v$. Now, let τ_1 and τ_2 belong to T_v. If $\vdash v : \tau_1 \vee \tau_2$, then trivially $\tau_1 \vee \tau_2 \in T_v$. Otherwise, by *(subsum)*, $\nvdash v : \tau_1$ and $\nvdash v : \tau_2$. Since τ_1 and τ_2 belong to T_v, then $\vdash v : \neg\tau_1$ and $\vdash v : \neg\tau_2$. Then, Lemma 7 entails $\vdash v : \neg\tau_1 \wedge \neg\tau_2$ and $\neg\tau_1 \wedge \neg\tau_2 \simeq \neg(\tau_1 \vee \tau_2)$. By Lemma 6, $\vdash v : \neg(\tau_1 \vee \tau_2)$; so, $\neg(\tau_1 \vee \tau_2) \in T_v$. By closure under $\neg$, this entails that $\tau_1 \vee \tau_2 \in T_v$. Finally, closure under $\wedge$ easily follows from closure under $\neg$ and $\vee$.

We now show that $T_v = \mathcal{T}$, i.e., that, for every $\tau \in \mathcal{T}$, it holds that $\tau \in T_v$. For what we have just shown, we can only consider the case when τ is an atomic type t. There are two cases to consider:

1. Let us first assume that v and t are not compatible and prove that $\vdash v : \neg t$. Since $v \in \mathcal{V}$, it is typeable and, hence, there exists a type derivation for v. Without loss of generality, such a derivation ends with a (possible useless) instance of *(subsum)* preceded by an instance of *(const)/(new)/(abstr)*, according to the kind of v; the latter yields $\vdash v : \tau'$. By definition of the typing rules, τ' is

   - a basic type, if v is a constant;
   - a record type in conjunction with some negated record types, if v is an object creation;
   - a conjunction of arrow types and of negated arrow types, if v is an abstraction.

   In the first case, $[\![\tau']\!]_{\mathcal{B}} \subseteq \mathcal{K}$. Since t is not compatible with v, it cannot be a basic type; hence, $[\![t]\!]_{\mathcal{B}} \subseteq \mathcal{B} \setminus \mathcal{K}$. Thus, $[\![\neg t]\!]_{\mathcal{B}} \supseteq \mathcal{K}$ and, consequently, $\tau' \leq \neg t$. By *(subsum)*, we easily conclude that $\vdash v : \neg t$. In the second and third case, we reason in a similar way but we replace $\mathcal{K}$ with $\mathcal{P}_f(\mathcal{L} \times \mathcal{B})$ and $\mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega)$, respectively.

2. Let us then assume that v and t are compatible and reason on the kind of v:

   $v = c$**:** by *(const)*, $\vdash c : \mathbb{B}_c$. By definition, $\mathbb{E}(\mathbb{B}_c) = Val_{\mathbb{B}_c} = \{c\}$; hence, $\mathbb{B}_c \leq t$ or $\mathbb{B}_c \leq \neg t$, according to whether $c \in \mathbb{E}(t)$ or not. By rule *(subsum)*, $\vdash c : t$ or $\vdash c : \neg t$.

   $v =$ **new** $C(\widetilde{u})$**:** in this case, $t = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$. By rule *(new)* and *(subsum)*, $\vdash v : t$ if $type(C) \leq [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ (indeed, since v belongs to $\mathcal{V}$, it must be typeable and, hence, $\vdash \widetilde{u} : \widetilde{\alpha'}$, where $\widetilde{\alpha'}$ is the sequence of types of the fields in C). Otherwise, since $type(C) \not\leq [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, we have that $type(C) \wedge \neg[\widetilde{a : \alpha}, \widetilde{m : \mu}] \neq \mathbf{0}$. Hence, we can use the latter type in *(abstr)* and then *(subsum)* to infer that $\vdash v : \neg t$.

   $v = \lambda_{(m,\rho)} x.e$**:** in this case, $t = \alpha \to \alpha'$. By rule *(abstr)* and *(subsum)*, $\vdash v : t$ if $\rho(m) = \bigwedge_{i \in I}(\alpha_i \to \alpha'_i) \leq \alpha \to \alpha'$ (indeed, since v belongs to $\mathcal{V}$, it must be typeable and, hence, for all $i \in I$, it holds that $x : \alpha_i \vdash e : \alpha'_i$). Otherwise, we reason like in the previous case with type $\bigwedge_{i \in I}(\alpha_i \to \alpha'_i) \wedge \neg(\alpha \to \alpha')$□

Lemma 9. $[\![\tau_1 \vee \tau_2]\!]_{\mathcal{V}} = [\![\tau_1]\!]_{\mathcal{V}} \cup [\![\tau_2]\!]_{\mathcal{V}}$.

Proof. By Lemma 8, 7 and 6, we have that $[\![\tau_1 \vee \tau_2]\!]_{\mathcal{V}} = [\![\neg((\neg\tau_1) \wedge (\neg\tau_2))]\!]_{\mathcal{V}} = \mathcal{V} \setminus ([\![\neg\tau_1]\!]_{\mathcal{V}} \cap [\![\neg\tau_2]\!]_{\mathcal{V}}) = \mathcal{V} \setminus ((\mathcal{V} \setminus [\![\tau_1]\!]_{\mathcal{V}}) \cap (\mathcal{V} \setminus [\![\tau_2]\!]_{\mathcal{V}})) = \mathcal{V} \setminus (\mathcal{V} \setminus ([\![\tau_1]\!]_{\mathcal{V}} \cup [\![\tau_2]\!]_{\mathcal{V}})) = [\![\tau_1]\!]_{\mathcal{V}} \cup [\![\tau_2]\!]_{\mathcal{V}}$. □

Proposition 2. $[\![\cdot]\!]_{\mathcal{V}}$ *is a set-theoretic interpretation.*

Proof. By Lemma 5, 7 and 8. □

C Closing the Circle

We are now ready to prove Theorem 1. To this aim, we first need a preliminary technical Lemma on the bootstrap model.

Lemma 10. *Let* $d = \{(l'_1 : d_1), \ldots, (l'_n : d_n)\} \in [\![\bigwedge_{\rho\in P}\rho \setminus \bigvee_{\rho\in N}\rho]\!]$. *Then,*

$$d \in \left[\!\!\left[[l'_1 : \bigwedge_{\rho\in P}(\rho \uparrow l'_1) \setminus \bigvee_{\rho\in N}(\rho \downarrow l'_1)\ ,\ \cdots\ ,\ l'_n : \bigwedge_{\rho\in P}(\rho \uparrow l'_n) \setminus \bigvee_{\rho\in N}(\rho \downarrow l'_n)] \right]\!\!\right]$$

where

$$\rho \uparrow l = \begin{cases} \rho(l) & \text{if } l \in dom(\rho) \\ \mathbf{1} & \text{otherwise} \end{cases} \qquad \rho \downarrow l = \begin{cases} \rho(l) & \text{if } l \in dom(\rho) \subseteq \{l'_1, \ldots, l'_n\} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

and $dom(\rho)$ *denotes* $\{l_1, \ldots, l_m\}$ *whenever* $\rho = [l_1 : \tau_1, \ldots, l_m : \tau_m]$.

Proof. Because the bootstrap model is set-theoretic (see Prop. 1), we have that

$$\left[\!\!\left[\bigwedge_{\rho\in P}\rho \setminus \bigvee_{\rho\in N}\rho\right]\!\!\right] = \bigcap_{\rho\in P}[\![\rho]\!] \setminus \bigcup_{\rho\in N}[\![\rho]\!]$$

and hence

- for all $\rho \in P$, it holds that $d \in [\![\rho]\!]$; and
- for all $\rho \in N$, it holds that $d \notin [\![\rho]\!]$.

From the first item, $d \in [\![\rho]\!]$ if and only if $dom(\rho) \subseteq \{l'_1, \ldots, l'_n\}$, and, for all $l \in dom(\rho)$, it holds that $d_i \in [\![\rho(l)]\!]$, where $l'_i = l$. Thus, for every $i = 1, \ldots, n$, we have that

$$d_i \in \bigcap_{\rho\in P\,:\,l'_i\in dom(\rho)} [\![\rho(l'_i)]\!] = \bigcap_{\rho\in P} [\![\rho \uparrow l'_i]\!] = \left[\!\!\left[\bigwedge_{\rho\in P}(\rho \uparrow l'_i)\right]\!\!\right]$$

Similarly, from the second item, $d \notin [\![\rho]\!]$ if and only if either $dom(\rho) \not\subseteq \{l'_1, \ldots, l'_n\}$ or there exists $l \in dom(\rho)$ such that $l'_i = l$ but $d_i \notin [\![\rho(l)]\!]$. Thus, for every $i = 1, \ldots, n$, we have that

$$d_i \notin \bigcup_{\rho\in N\,:\,l'_i\in dom(\rho)\subseteq\{l'_1,\ldots,l'_n\}} [\![\rho(l'_i)]\!] = \bigcup_{\rho\in N} [\![\rho \downarrow l'_i]\!] = \left[\!\!\left[\bigvee_{\rho\in N}(\rho \downarrow l'_i)\right]\!\!\right]$$

So, we have proved that $d_i \in [\![\bigwedge_{\rho\in P}(\rho \uparrow l'_i) \setminus \bigvee_{\rho\in N}(\rho \downarrow l'_i)]\!]$, for every $i = 1, \ldots, n$. This entails the thesis. □

Notice that, in the definition of $\rho \uparrow l$, the condition $dom(\rho) \subseteq \{l'_1, \ldots, l'_n\}$ would be redundant, since it follows by the fact that $\rho \in P$ and $d \in \bigcap_{\rho\in P}[\![\rho]\!]$. By contrast, there may be a $\rho \in N$ such that $dom(\rho) \not\subseteq \{l'_1, \ldots, l'_n\}$ (and this is the reason for d not being in $[\![\rho]\!]$) but with some $l'_i \in dom(\rho)$. We cannot consider such a ρ in the overall union for d_i because, in principle, it could be $d_i \in [\![\rho(l'_i)]\!]$. This is the reason for having the condition $dom(\rho) \subseteq \{l'_1, \ldots, l'_n\}$ in the latter union of the proof and, consequently, in the definition of $\rho \downarrow l$.

Proof of Theorem 1 It suffices to prove, for every τ, that

$$[\![\tau]\!]_{\mathcal{V}} = \emptyset \Longleftrightarrow [\![\tau]\!] = \emptyset$$

Indeed, $\tau_1 \leq \tau_2$ if and only if $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$ and this holds if and only if $[\![\tau_1]\!] \setminus [\![\tau_2]\!] = [\![\tau_1 \backslash \tau_2]\!] = \emptyset$. And similarly for $[\![\cdot]\!]_{\mathcal{V}}$ and $\leq_{\mathcal{V}}$, thanks to Proposition 2.

($\Leftarrow$) Because of Proposition 1, we have that $[\![\tau]\!] = \emptyset$ if and only if $\tau \simeq \mathbf{0}$. Thus, we can easily conclude by Lemmata 6 and 5.

($\Rightarrow$) We prove the contrapositive, i.e.

$$d \in [\![\tau]\!] \Longrightarrow [\![\tau]\!]_{\mathcal{V}} \neq \emptyset$$

The proof is by structural induction on $d \in D$. Notice that this induction can be done because the bootstrap model is structural (see Proposition 1); indeed, by using the $\gg$ relation, we have that the induction is mathematically well-founded.

We have three cases for d, viz. whether it belongs to $\mathcal{K}$, to $\mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega)$ or to $\mathcal{P}_f(\mathcal{L} \times \mathcal{B})$:

1. $d \in \mathcal{K}$: then $d \in Val_{\mathbb{B}}$, for some $\mathbb{B}$; hence, $\mathbb{E}(\mathbb{B}_d) = \{d\} \subseteq \mathbb{E}(\tau)$ that implies $\mathbb{B}_d \leq \tau$ (because $[\![\cdot]\!]$ is well-founded); moreover, $\vdash d : \mathbb{B}_d$, by rule (*const*). We can conclude that $\vdash d : \tau$, by using rule (*subsum*). Hence, $d \in [\![\tau]\!]_{\mathcal{V}}$.

2. $d \in \mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega)$: then, $d = \{(d_1, d'_1), \cdots, (d_n, d'_n)\}$, and it belongs to the set

$$[\![\tau]\!] \cap \mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega) = \bigcup_{(P,N) \in \mathcal{N}(\tau)} \left(\mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega) \cap \left(\bigcap_{t \in P} [\![t]\!] \setminus \bigcup_{t \in N} [\![t]\!] \right) \right)$$

   Thus, we can write

$$\tau = \bigvee_{(P,N) \in \mathcal{N}(\tau)} \tau_{(P,N)}$$

   where, for every pair $(P, N) \in \mathcal{N}(\tau)$, we have that $P = \{\alpha_1 \to \alpha'_1, \ldots, \alpha_n \to \alpha'_n\}$, $N \cap \mathbb{T}_{\mathbf{fun}} = \{\alpha_{n+1} \to \alpha'_{n+1}, \ldots, \alpha_m \to \alpha'_m\}$ and

$$\tau_{(P,N)} = \left(\bigwedge_{i=1\ldots n} \alpha_i \to \alpha'_i \right) \setminus \left(\bigvee_{j=n+1\ldots m} \alpha_j \to \alpha'_j \right)$$

   Hence, there exists $\tau_{(P,N)} \not\simeq \mathbf{0}$, since by hypothesis $\tau \not\simeq \mathbf{0}$. Moreover, $[\![\tau]\!]_{\mathcal{V}} = \bigcup_{(P,N) \in \mathcal{N}(\tau)} [\![\tau_{(P,N)}]\!]_{\mathcal{V}}$. To conclude the desired $[\![\tau]\!]_{\mathcal{V}} \neq \emptyset$, it suffices to prove that $[\![\tau_{(P,N)}]\!]_{\mathcal{V}} \neq \emptyset$. Since $\tau_{(P,N)} \not\simeq \mathbf{0}$, by rule *(abstr)* we can conclude if we find a proper $\lambda_{(m,\rho)} x.e$ such that $\rho(m) = \bigwedge_{i=1,..,n}(\alpha_i \to \alpha'_i)$, and, for all $i = 1, .., n$, it holds that $x : \alpha_i \vdash e : \alpha'_i$. It can be easily verified that a possible choice is to have $\rho = [m : \bigwedge_{i=1,..,n}(\alpha_i \to \alpha'_i)]$ and $e = (\mathbf{new}\ C_\rho).m(x)$; indeed, $\bigwedge_{i=1,..,n}(\alpha_i \to \alpha'_i) \simeq (\bigvee_{i=1,..,n} \alpha_i) \to (\bigwedge_{i=1,..,n} \alpha'_i)$.

3. $d \in \mathcal{P}_f(\mathcal{L} \times \mathcal{B})$: thus, $d = \{(l'_1, d_1), \ldots, (l'_n, d_n)\}$, and it belongs to the set

$$[\![\tau]\!] \cap \mathcal{P}_f(\mathcal{L} \times \mathcal{B}) = \bigcup_{(P,N) \in \mathcal{N}(\tau)} \left(\mathcal{P}_f(\mathcal{L} \times \mathcal{B}) \cap \left(\bigcap_{t \in P} [\![t]\!] \setminus \bigcup_{t \in N} [\![t]\!] \right) \right)$$

   Hence, we can find a pair $(P, N) \in \mathcal{N}(\tau)$ such that $d \in \mathcal{P}_f(L \times \mathcal{B}) \cap (\bigcap_{t \in P} [\![t]\!] \setminus \bigcup_{t \in N} [\![t]\!])$. Notice that, if t is an atom different from a record type,

then $\mathcal{P}_f(\mathcal{L} \times \mathcal{B}) \cap [\![t]\!] = \varnothing$ (recall that $[\![\cdot]\!]$ is structural). Thus, it suffices to consider $P \subseteq \mathbb{T}_{\mathbf{rec}}$; so, we have

$$d \in \bigcap_{\rho \in P} [\![\rho]\!] \setminus \bigcup_{\rho \in N} [\![\rho]\!]$$

Because of Lemma 10, $\{(l'_1, d_1), \ldots, (l'_n, d_n)\} \in [\![[l'_1 : \tau_1, \ldots, l'_n : \tau_n]]\!]$, where $\tau_i = \bigwedge_{\rho \in P}(\rho \uparrow l'_i) \setminus \bigvee_{\rho \in N}(\rho \downarrow l'_i)$. By definition of the bootstrap model, this means that, for every $i = 1, \ldots, n$, it holds that $d_i \in [\![\tau_i]\!]$. We can now use the inductive hypothesis applied to the latter judgements and obtain that $[\![\tau_1]\!]_{\mathcal{V}} \neq \varnothing, \ldots, [\![\tau_n]\!]_{\mathcal{V}} \neq \varnothing$; consequently, also $[\![[l'_1 : \tau_1, \ldots, l'_n : \tau_n]]\!]_{\mathcal{V}} \neq \varnothing$. To conclude, we use Lemma 9 and obtain that $[\![[l'_1 : \tau_1, \ldots, l'_n : \tau_n]]\!]_{\mathcal{V}} \subseteq [\![\tau]\!]_{\mathcal{V}}$, because τ is a disjunction of types ($\bigcup_{(P,N) \in \mathcal{N}(\tau)}$) and we took one of its disjuncts (P, N). □