Semantic Subtyping for Objects and Classes

Ornela Dardha[1] Daniele Gorla[2] Daniele Varacca[3]

[1] INRIA Focus Team / Università di Bologna (Italy)
[2] Dip. di Informatica, "Sapienza" Università di Roma (Italy)
[3] PPS - Université Paris Diderot & CNRS (France)

Abstract. 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 to 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 the framework can also accommodate the nominal subtyping. Several examples show the expressivity and the practical advantages of our proposal.

1 Introduction

Type systems for programming languages are often based on a subtyping relation on types. There are two main approaches for defining the subtyping relation: the *syntactic* approach and the *semantic* one. The syntactic approach is more common: the subtyping relation is defined by means of a formal system of deductive rules. One proceeds as follows: first define the language, then the set of syntactic types and finally the subtyping relation by inference rules. In the semantic approach, instead, one starts from a model of the language and an interpretation of types as subsets of the model. The subtyping relation is then defined as inclusion of sets denoting types. This approach has received less attention than the syntactic one as it is more technical and constraining: it is not trivial to define the interpretation of types as subsets of a model. On the other hand, it presents several advantages: it allows a natural definition of boolean operators, also the meaning of types is more intuitive for the programmer, who need not be aware of the theory behind the curtain.

The first use of the semantic approach goes back to two decades ago [3,10]. More recently, Hosoya and Pierce [15,16] 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. Castagna et al [8,13] 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. This approach can also be applied to π-calculus [25]. Castagna et

al. [9] defined the $\mathbb{C}\pi$ language, a variant of the asynchronous π-calculus where channel types are augmented with boolean connectives. Finally, semantic subtyping is adopted in a flow-typing calculus [23]. 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.

In the present paper, we address the semantic subtyping approach by applying it to an object-oriented core language. Our starting point is *Featherweight Java* (FJ) [17], which is a functional fragment of Java. From a technical point of view the development is challenging. It follows [13], but with the difference that we do not have higher-order values. Therefore, we cannot directly reuse their results. Instead, we define from scratch the semantic model that induces the subtyping relation, and we prove some important theoretical results. The mathematical technicalities, however, are transparent to the final user. Thus, the overheads are hidden to the programmer. The benefits reside in that the programmer now has a language with no additional complexity (w.r.t. standard Java) but with an easier-to-write, more expressive set of types. There are several other reasons and benefits that make semantic subtyping very appealing in an object-oriented setting. For example, it allows us to very easily handle powerful *boolean type constructors* and model both *structural* and *nominal* subtyping. 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 modeling multimethods. Featherweight Java [17] 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 [5,7]; according to [6], 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 {
    ...
    int length (string s){ ... }
}
```

```
class B extends A {
    ...
    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 can be used by the compiler for typechecking terms of a language. But they can also be used directly by the programmer. 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 Section 5.3). 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 $\neg \mathit{ValuedObject}$.

While the example is quite simple, we believe it exemplifies the 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 the classical solution offered by an object-oriented paradigm.

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 programmers are used to nominal subtyping, but other languages [12,14,18,19,20,22,24] 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: a class A is a subtype of a class B if and

[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.

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. 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 [14,19]. In this paper we follow the reverse direction and give another contribution. The definition of structural subtyping as inclusion of sets fits perfectly the definition of semantic subtyping. So, we integrate both approaches in the same framework. In addition to that, with minor modifications, it is also possible to include in the framework the choice of using nominal subtyping without changing the underlying theory. Thus, since both nominal and structural subtyping are thoroughly used and have their benefits, in our work, we can have them both and so it becomes a programmer's decision on what subtyping to adopt.

Plan of the paper. In Section 2 we present the syntax of types and terms. In Section 3 we define type models, semantic subtyping relation and present also the typing rules. In Section 4 we present the operational semantics and the soundness of the type system. Proofs of theorems and auxiliary lemmas can be found in [11]. Section 5 gives a discussion on the calculus and Section 6 concludes the paper.

2 The calculus

In this section, we present the syntax of the calculus, starting first with the types and then 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:

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

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, viz. μ. Type $\mathbf{0}$ denotes the empty type. Type $\mathbb{B}$ denotes the basic types: integers, booleans, void, etc. Type $[\widetilde{l : \tau}]$ denotes a record type, 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 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 Section 5.2. 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.

Definition 1 (Types). *The* pre-types *are the regular trees (i.e., the trees with a finite number of non-isomorphic subtrees) produced by the syntax of type terms.*

The set of types*, denoted by $\mathcal{T}$, is the largest set of* well-formed *pre-types, i.e. the ones for which 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.

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, which leads us to restrict them to the regular and the well-formed ones. Indeed, if a tree is not-regular, then it is difficult to write it down in a finite way; since we want our types to be usable in practice, we require regular trees that can be easily written down, 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. The restriction to well-formed types is required to avoid meaningless types; the same choice is used in [13]. A different restriction, called *contractiveness*, is used for instance in [4], where non-regular types are also allowed.

2.2 Terms

Our calculus is based on FJ [17] rather than, for example, the object-oriented calculus in [1], 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 [17], apart from the absence of the *cast* construct and the presence of the **rnd** primitive. We have left out the first construct for the sake of simplicity; it is orthogonal to the aim of the current work and it can be added to the language without any major issue. The second construct is a nondeterministic choice operator. This operator is technically necessary to obtain a completeness result. Indeed, we interpret method types not as function but as relations, following the same line of [13]; thus, a nondeterministic construct is needed to account for this feature. In addition, **rnd** can be used to model side-effects. 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	$::= C\ (\widetilde{\beta\ b}; \widetilde{\alpha\ a})\ \{\ \textbf{super}(\widetilde{b});\ \widetilde{\textbf{this}.a = a};\ \}$
Method declaration	M	$::= \alpha\ m\ (\alpha\ a)\ \{\ \textbf{return}\ e;\ \}$
Expressions	e	$::= x \mid c \mid e.a \mid e.m(e) \mid \textbf{new}\ C(\widetilde{e}) \mid \textbf{rnd}(\alpha)$

A *program* is a pair $(\widetilde{L}, e)$ consisting of a sequence of class declarations (inducing a class hierarchy, as specified by the inheritance relation) $\widetilde{L}$ where the expression e is evaluated. 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. For simplicity, we use unary methods without compromising 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 for having as fields all the arguments needed. On the other hand, we exploit this simplification in the theoretical development of our framework. Finally, expressions e are variables, constants, field accesses, method invocations, object creations and random choices among values of a given type. 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 to focus our attention on the new features of our framework. The same sanity checks are assumed also in FJ [17].

3 Semantic subtyping

3.1 Models

Having defined the raw syntax, we should now introduce the typing rules. They would typically involve a subsumption rule, that invokes 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 values of type τ_1 is a subset of the set of 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 judgments for values; but the typing rules use the subtyping relation which we are going to define. So, there is a circularity. To break this circle, we follow the path of [13] 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 value, 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

- $type(Object) = [\,]$;
- $type(C) = \rho$, 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 $\alpha \to \beta$, then $\rho(m) = \alpha \to \beta$;
    - if $\rho'(m)$ is defined and $m \notin C$, then $\rho(m) = \rho'(m)$;
    - if $\rho'(m) = \bigwedge_{i=1}^{n} \alpha_i \to \beta_i$, $m \in C$ with type $\alpha \to \beta$ and $\mu = \alpha \to \beta \wedge \bigwedge_{i=1}^{n} \alpha_i \setminus \alpha \to \beta_i \leq \rho'(m)$, then $\rho(m) = \mu$.

  $type(C)$ is undefined, otherwise.

Table 1. Definition of function $type(\_)$

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) and should capture the meaning of type constructors. Notice that there can be several models and it is not guaranteed that they all induces the same subtyping relation. For our purposes, we only need to find one suitable model that we shall call *bootstrap model* $[\![\_]\!]_{\mathcal{B}}$. The construction of this model is beyond the scope of this paper, and for a detailed presentation we refer the reader to [11]. Then, set inclusion in the bootstrap model induces a subtyping relation: $\tau_1 \leq_{\mathcal{B}} \tau_2 \iff [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$.

3.2 Typing Terms

In the typing rules for our language we use the subtyping relation just defined to derive typing judgments $\Gamma \vdash_{\mathcal{B}} e : \tau$. 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.

Table 1 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. 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 line as [13] 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.

```
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 × i}
    real G() {return this.F()}
}
```

As usual **int** $\leq$ **real** $\leq$ **compl**. At run time, the function G returns a **complex** number, instead of a **real**. 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 2. 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 assume that, for any basic type $\mathbb{B}$, there exists a fixed set of constants $Val_{\mathbb{B}} \subseteq \mathcal{K}$ such that the elements of this set have type $\mathbb{B}$. Notice that, for any two basic types $\mathbb{B}_1$ and $\mathbb{B}_2$, the sets $Val_{\mathbb{B}_i}$ may have a non empty intersection. Rule (*var*) derives that x has type α, if x is assigned type α in Γ. Let us now concentrate on rules (*field*) and (*m-inv*). 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. Moreover, like in [13], 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 **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. This property guarantees that the interpretation of types as sets of values induces the same subtyping relation as the bootstrap model. Rule (*rnd*) states that **rnd**(α) is of type α. 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.

Typing Expressions :

$$(subsum)\ \frac{\Gamma \vdash e : \alpha_1 \quad \alpha_1 \leq \alpha_2}{\Gamma \vdash e : \alpha_2} \qquad (const)\ \frac{c \in Val_{\mathbb{B}}}{\Gamma \vdash c : \mathbb{B}}$$

$$(var)\ \frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha} \qquad (field)\ \frac{\Gamma \vdash e : [a : \alpha]}{\Gamma \vdash e.a : \alpha}$$

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

$$(new)\ \frac{type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}] \qquad \Gamma \vdash \widetilde{e} : \widetilde{\beta} \qquad \widetilde{\beta} \leq \widetilde{\alpha} \qquad \rho = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \qquad \rho \not\simeq \mathbf{0}}{\Gamma \vdash \mathbf{new}\ C(\widetilde{e}) : \rho}$$

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 :

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

Typing Programs :

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

Table 2. Typing Rules

3.3 Closing the circle

To close the circle, one should now interpret types as sets of values. In our calculus, a natural notion of value includes the constants and the objects initialized by only 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. [13].

To overcome this problem we use the more general notion of *pseudo-value.* A pseudo-value is a closed, well-typed expression that cannot reduce further. The interpretation of an α-type is the set of pseudo-values of that type For μ-types, we interpret an arrow type as a set of pairs (α, w) such that it is possible to assign to the normal form w the return type of μ whenever the input argument of the method is assigned input type of μ i.e., type α, which "closes" the normal form w. The details can be found in [11].

$$\textit{(f-ax)}\ \frac{type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]}{(\mathbf{new}\ C(\widetilde{u})).a_i \to u_i} \qquad \textit{(f-red)}\ \frac{e \to e'}{e.a \to e'.a}$$

$$\textit{(r-ax)}\ \frac{\vdash e : \alpha}{\mathbf{rnd}(\alpha) \to e} \qquad \textit{(m-ax)}\ \frac{body(m, u, C) = \lambda x.e}{(\mathbf{new}\ C(\widetilde{u'})).m(u) \to e[{}^{u}/_{x}, {}^{\mathbf{new}\ C(\widetilde{u'})}/_{\mathbf{this}}]}$$

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

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

Table 3. Operational semantics

Using the above intuitions, we define the interpretation function $[\![\cdot]\!]_{\mathcal{V}}$ and, consequently, the subtyping relation $\leq_{\mathcal{V}}$. A priori, the new relation $\leq_{\mathcal{V}}$ could 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. Hence, we can prove the following result, the proof of which can be found in [11].

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

4 Operational Semantics and Soundness of the Type System

The operational semantics is defined through the transition rules of Table 3; these are essentially the same as in FJ. There are only two notable differences: we use function *type* to extract the fields of an object, instead of defining an ad-hoc function; 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 in the following and the if cases are to be considered in order:

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

Notice that method resolution is performed at runtime, by keeping into account the *dynamic* type of the argument; this is called *multimethods* and is different from what happens in Java, where method resolution is performed at compile time by keeping into

account the *static* type of the argument. A more traditional modeling of overloading is possible and easy to model.

Soundness of the Type System. Theorem 1 does not automatically imply that the definitions put forward in Sections 3 and 4 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. The full proofs can be found in [11].

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

Proof. The proof is by induction on the length of $e \to e'$.

Theorem 3 (Progress). *If* $\vdash e : \alpha$ *where 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.

5 Discussion on the calculus

5.1 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 cannot be done. 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:

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

α being the solution of the recursive type equation $\alpha = [val : \textbf{int}, succ : (\alpha \vee \textbf{unit})]$. Now, we can create the list $\langle 1, 2 \rangle$ by writing the value **new** $intList(1, \textbf{new}\ intList(2, \textbf{null}))$.

5.2 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 twice a multimethod m within class A:

$$
\begin{array}{l}
\textbf{class}\ A\ \textbf{extends}\ Object\ \{ \\
\qquad \ldots \\
\qquad \alpha_1\ m\ (\beta_1\ x)\{\textbf{return}\ e_1\} \\
\qquad \alpha_2\ m\ (\beta_2\ x)\{\textbf{return}\ e_2\} \\
\}
\end{array}
$$

We then replace it with the following declarations:

$$\begin{array}{ll}
\textbf{class } A1 \textbf{ extends } Object \{ & \textbf{class } A \textbf{ extends } A1 \{ \\
\quad \ldots & \quad \ldots \\
\quad \alpha_1\ m\ (\beta_1\ x)\{\textbf{return } e_1\} & \quad \alpha_2\ m\ (\beta_2\ x)\{\textbf{return } e_2\} \\
\} & \}
\end{array}$$

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 Table 1) 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.

5.3 Implementing Typical Java-like Constructs

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

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

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 [13] 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*:

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

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*()).$m_\alpha(e)$.

5.4 Nominal subtyping vs. Structural subtyping

The semantic subtyping is a way to allow programmers use powerful typing disciplines, but we do not want to bother them with the task of explicitly writing structural types. 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 Section 5.1. 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 Section 5.1.

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.

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.

It remains to describe how we can use nominal subtyping in place of the 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. Thus, if we consider the following class declarations

$$\begin{array}{c}\textbf{class } C \textbf{ extends } \mathit{Object}\ \{\ \} \\ \textbf{class } D \textbf{ extends } \mathit{Object}\ \{\ \}\end{array}$$

the expressions $(\textbf{new } A()).m(\textbf{new } C(), \textbf{new } C())$, $(\textbf{new } A()).m(\textbf{new } D(), \textbf{new } C())$ and $(\textbf{new } A()).m(\textbf{new } \mathit{Object}(), \textbf{new } C())$ typecheck, whereas $(\textbf{new } A()).m(\textbf{new } C(), \textbf{new } D())$ and $(\textbf{new } A()).m(\textbf{new } C(), \textbf{new } \mathit{Object}())$ do not.

6 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 [13]. A preliminary formal development can be found in the first author's M.S thesis [11]. These are intermediate steps towards a prototype programming environment where writing and evaluating the performances of code written in the new formalism.

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 choice made in the implementation of $\mathbb{C}$Duce. The fact that we have assumed nondeterministic methods can also help in modeling a mutable state: as we have said, the input-output behavior of a function can be seen as nondeterministic since, besides its input, the function has access to the state.

Another possibility for enhancing the language is the introduction of higher-order values, in the same vein as the Scala programming language [21]; since the framework of [13] is designed for a higher-order language, the theoretical machinery developed therein should be easily adapted to the new formalism.

References

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

2. R. Agrawal, L.G. de Michiel and B. G. Lindsay. Static type checking of multimethods. In *Proc. of OOPSLA*, pages 113–128. ACM Press, 1991.
3. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proc. of FPCA*, pages 31–41. ACM, 1993.
4. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *Proc. of ECOOP09*, pages 2–26. Springer, 2009.
5. 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.
6. J.T. Boyland and G. Castagna. Parasitic Methods: an implementation of multi-methods for Java. In *Proc. of OOPSLA*. ACM Press, 1997.
7. G. Castagna. *Object-oriented programming: a unified foundation*. Progress in Theoretical Computer Science series, Birkäuser, Boston 1997.
8. G. Castagna. Semantic subtyping: Challenges, perspectives, and open problems. In *Proc. of ICTCS*, pages 1–20, 2005.
9. G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the pi-calculus. *Theoretical Computer Science*, 398(1-3):217–242, 2008.
10. F. M. Damm. Subtyping with union types, intersection types and recursive types. In *Proc. of TACS*, pages 687–706. Springer, 1994.
11. O. Dardha. Sottotipaggio semantico per linguaggi ad oggetti. MS thesis, Dip. Informatica, "Sapienza" Univ. di Roma. Available online at `www.dsi.uniroma1.it/~gorla/TesiDardha.pdf`.
12. 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.
13. 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.
14. J. Gil and I. Maman. Whiteoak: introducing structural typing into Java. In *Proc. of OOPSLA*, pages 73–90. ACM, 2008.
15. H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *SIGPLAN Notices*, 36(3):67–80, 2001.
16. H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
17. 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.
18. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.11*, 2008.
19. D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proc. of ECOOP*, pages 260–284. Springer, 2008.
20. D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In *Proc. of ESOP*, pages 95–111. Springer, 2009.
21. M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, 2004.
22. K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.
23. D. Pearce. *Sound and Complete Flow Typing with Unions, Intersections and Negations*, In Proc. of VMCAI, pages 335–354, 2013.
24. D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. of POPL*, pages 40–53. ACM, 1997.
25. D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2003.