Semantic Subtyping for Objects and Classes

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

[1] Dip. Scienze dell'Informazione, 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 approaches. Semantic subtyping is an approach to defining the 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, the final user 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 structural subtyping, we show how the approach can accommodate the nominal style of 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.

The semantic 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 the types as subsets of a model. However, it presents several advantages: for instance, it allows a natural definition of boolean operators. Also the meaning of the types is more intuitive for the programmer, who can also be unaware of the theory behind the curtain.

The first use of the semantic approach goes back to two decades ago [3,12]. More recently, Hosoya and Pierce have adopted this approach in [17,18,19] to define XDuce, an XML-oriented language designed specifically to transform XML documents into other XML documents satisfying certain properties. The values of this language are fragments of XML documents; types are interpreted as sets of documents, more precisely as sets of values. The subtyping relation is established as inclusion of these sets. The type system contains boolean types,

product types and recursive types. There are no function types and no functions in the language.

Castagna et al. in [9,11,15] extend the XDuce language with first-class functions and arrow types; this yields a higher-order language, named $\mathbb{C}$Duce, 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.

The semantic approach can also be applied to the π-calculus [24,28]. Castagna, Varacca and De Nicola in [10] have used this technique to define the $\mathbb{C}\pi$ language, a variant of the asynchronous π-calculus, where channel types are augmented with boolean connectives interpreted in an obvious way.

We aim at applying the semantic subtyping approach to an object-oriented core language. Our starting point is the language *Featherweight Java* [20], which is a functional fragment of Java. From a technical point of view, our development follows [15], with the important 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, who sees 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 that make the 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 [7]. 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 [20] is a minimal 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 [5,8]; 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){ //returns the number of characters in s }
}
class B extends A {
        ...
        int length(int n){ //returns the the number of digits in n }
}
```

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 to $(\mathbf{string} \vee \mathbf{int}) \to \mathbf{int}$.

The use of negation types. As we will see, negation types are very useful to the compiler for typing terms of language. But they can be also useful directly to 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 Euros. 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 case, the type of objects that have the method *getValue* is denoted by

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

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

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

that takes in input only objects that do not have a price, i.e. a method named *getValue* (the return type α does not play any role here).

One could argue that it is difficult to statically know that an object does not have the method *getValue* and thus no reasonable application of the method *barter* can be well-typed. However, it is not difficult to explicitly build a collection of objects that do not have the method *getValue*, by dynamically checking the presence of the method. This is possible thanks to the **instanceof** construction (described in Section 6.3). The method *barter* can now be applied to any object of that list, and the application will be well typed.

[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.

[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.

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, 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.

Structural subtyping. An orthogonal issue, typical of object-oriented languages, is the *nominal* vs. *structural* subtyping question. In a language where the subtyping is nominal, A is a subtype of B if and only if it is declared to be so, that is if the class A extends (or implements) the class (or interface) B; these relations must be declared by the programmer and are based on the names of the classes and interfaces concerned. Java programmers are used to nominal subtyping, but other languages [14,16,21,22,23,26,27] are based on the structural approach. In this approach, the 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 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. The syntactic subtyping is more naturally linked to the nominal approach, though it can also be adapted to support the structural one, as shown in [16,22]. In this paper we follow the reverse direction. The definition of structural subtyping, as inclusion of sets corresponding to fields and methods of a class, fits perfectly 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.

Plan of the paper. The paper is organized as follows. In Section 2, we present the types and the language syntax. In Section 3, we define type models, used for the semantic subtyping. In Section 4, we define the subtyping relation in the semantic way, i.e. as inclusion between sets of values; in doing this, we also provide the typing rules for our language. In Section 5 we present the operational semantics and the soundness of the type system. In Section 6, we discuss some important issues: the possibility of programming recursive class definitions, of encoding standard multimethods, of implementing typical Java constructs and of integrating nominal typing and subtyping. We conclude in Section 7.

2 The calculus

In this section, we present the syntax of our calculus, i.e. the types and the language terms.

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). The arrow types are only needed to type the methods of our calculus. Since our language is first-order, i.e. methods are not first-class values, arrow types are introduced by a distinct syntactic category, viz. μ.

The type $\mathbf{0}$ denotes the empty type. The type $\mathbb{B}$ denotes the basic types: integers, reals, booleans, etc. The type $[\widetilde{l : \tau}]$ denotes a record type, where $\widetilde{\cdot}$ denotes a (possibly empty) sequence of elements of kind "$\cdot$". Thus, $\widetilde{l : \tau}$ 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 corresponding to the field a_i and $\rho(m_j)$ is the type corresponding to the method m_j. In each record type, we require that $a_i \neq a_j$ 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 6.2.

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.

First, notice that every finite tree obtained from the grammar of types is both regular and well-formed; so, it is a type. Problems can arise for infinite trees, and this fact leads us to restrict them to the regular and the well-formed ones. Indeed, if a tree is non-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; even worse, $\alpha = \neg\alpha$ 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 [15]. A different restriction, called *contractiveness*, is used for instance in [4], where non-regular types are also allowed.

2.2 Terms

Our language is based on *Featherweight Java* (FJ) [20], that is a minimal calculus based on Java. We have preferred this calculus w.r.t. [1] because of the widespread diffusion of Java. There is a correspondence between the calculus 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 [20], 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 can be added to the language without any problem: we just need to add the typing and reduction rules developed in [20], that of course rely on our semantic subtyping. The second construct is a nondeterministic choice operator; its name and role are the same as in [15]. Since method types will be interpreted as relations, we need some nondeterministic construct in the language to let all the machinery of the semantic subtyping work.

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. $\mathcal{C}$ will denote the set of constants of the language and we will use the meta-variable c to range over $\mathcal{C}$. 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 the following:

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

A *program* is a pair $(\widetilde{L}, e)$ formed by a sequence of class definitions (inducing a class hierarchy, as specified by the inheritance relation) $\widetilde{L}$ where the expression e is evaluated. Every class declaration L provides the name of the class, the name of the parent class, some field (each equipped with a type specification), one constructor K and some method declarations M. The constructor essentially initializes the fields of the object, by assigning values first to the fields inherited by the super-class and then 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 expressive power 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. 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.

3 Type 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 shall follow the path of [15] 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. Using this abstract notion of subtyping we can 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, the real subtyping relation coincides with the abstract one, and the circle is closed.

In this section we open the circle by presenting the notion of abstract model of types. The circle will be closed in the next section.

A model consists of a set D and an interpretation function $[\![\_]\!]_D : \mathcal{T} \to \mathcal{P}(D)$. Such a function shall 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, given an intuitive meaning of types, there may be several models that satisfy this requirement, and it is not guaranteed that they all induces the same subtyping relation. For our purposes, we only need to prove that there exists at least one suitable model that we shall call *bootstrap model*.

3.1 Set-theoretic interpretations and Models

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

Definition 1 *A* set-theoretic interpretation *of* $\mathcal{T}$ *is given by a set* D *and a function* $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ *such that, for any* $\tau_1, \tau_2, \tau \in \mathcal{T}$*, it holds that*

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

Notice that the above definition implies $[\![\tau_1 \vee \tau_2]\!] = [\![\tau_1]\!] \cup [\![\tau_2]\!]$, $[\![\tau_1 \backslash \tau_2]\!] = [\![\tau_1]\!] \setminus [\![\tau_2]\!]$ and $[\![\mathbf{1}]\!] = D$. Every set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ 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 emptiness, 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 define when an interpretation correctly represents the meaning of the type constructors. Notationally, for every basic type $\mathbb{B}$, we denote with $Val_{\mathbb{B}}$ the set of (basic) values of type $\mathbb{B}$. Moreover, we require that, for any 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 a record type $\rho = [\widetilde{l : \tau}]$, the intuition is that it should represent all the objects that have values of type τ_i in the field l_i, but that may have other fields as well.

Definition 2 *Given a record type* $[\widetilde{l : \tau}]$*, we define* $\widetilde{[l : [\![\tau]\!]]}$ *as:*

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

Intuitively, the interpretation of $[\widetilde{l : \tau}]$ is a set of relations $R \subseteq L \times D$ such that, whenever $(l, d) \in R$ and l is the i-th component of $\widetilde{l}$, d must belong to $[\![\tau_i]\!]$. Of course, since every field can usually assume several values (of a given type), we have to work with relations instead of functions, otherwise every field would be committed to a single value. Also, the record type $[a : \mathbf{0}]$ should be interpreted as the empty set, as our intuition suggests that we cannot instantiate any object of this type. Thus, we add the requirement that $dom(R) \supseteq \{\widetilde{l}\}$.

For a functional type $\alpha_1 \to \alpha_2$, the intuition is that it should represent the set of functions f such that, if d belongs to $[\![\alpha_1]\!]$, $f(d)$ must belong to $[\![\alpha_2]\!]$. For the same technical reasons explained in [15], we consider binary relations instead of functions: on the one hand, this simplifies the equations satisfied by the types; on the other hand, this is also necessary to model non-deterministic methods. In this paper, non-deterministic methods are represented by the choice operator **rnd**. Non-determinism becomes necessary when dealing with functions with side effects: depending on the state, the same input can produce different outputs, and this can be seen as non-deterministic from the point of view of the input-output behaviour.

Moreover, there is a notion of type error in the calculus: it is not possible to invoke an arbitrary method on an arbitrary argument. 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, whenever $(q, q') \in Q$ and $q \in [\![\alpha_1]\!]$, it holds that $q' \in [\![\alpha_2]\!]$.

Definition 3 *Let D be a set and X, Y subsets of D; we write D_Ω for $D \uplus \{\Omega\}$ and define:*

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

Notice that, if we replace D_Ω with D in the definition above, then $X \to Y$ would always be a subset of $D \to D$. This would imply that any arrow type would be a subtype of $\mathbf{1} \to \mathbf{1}$. If this were the case, 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 $D \subseteq X$. This result is true if we consider the covariance of arrow types.

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

Definition 4 *Let $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ 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 $\mathbb{E}D = \mathcal{C} \uplus \mathcal{P}(L \times D) \uplus \mathcal{P}(D \times D_\Omega)$) such that:*

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

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

Definition 5 *A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ 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. To construct such a value, we have to consider an infinite tree, that is excluded since values are the result of some computation. As a consequence, the type α does not contain values. Clearly this does not imply that all recursive types are trivial. In Section 6.2 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.

Definition 6 *A set-theoretic interpretation* $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ *is* structural *if:*

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

Definition 7 *A model* $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ *is* well-founded *if it induces the same subtyping relation as a structural set-theoretic interpretation.*

3.3 Bootstrap model

We now have to show that a well-founded model exists, and use it as the bootstrap model. To this aim, let us define $\mathcal{B}$ such that $\mathcal{B} = \mathbb{E}_f\mathcal{B}$, i.e. $\mathcal{B}$ is the solution of the equation $\mathcal{B} = \mathcal{C} \uplus \mathcal{P}_f(L \times \mathcal{B}) \uplus \mathcal{P}_f(\mathcal{B} \times \mathcal{B}_\Omega)$. In practice, it turns out that $\mathcal{B}$ is the set of finite terms generated by the following grammar:

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

We now define $[\![\tau]\!]_\mathcal{B} = \{d \in \mathcal{B} \mid d : \tau\}$, where judgment $d' : \tau$ is inductively defined as follows (it is assumed to be false in every non-depicted case):

$$\begin{aligned} c : \mathbb{B} &\text{ iff } c \in Val_\mathbb{B} \\ \{(l_1, d_1), \cdots, (l_n, d_n)\} : [l_1 : \tau_1, \cdots, l_n : \tau_n] &\text{ iff } \forall i.\ d_i : \tau_i \\ \{(d_1, d'_1), \cdots, (d_n, d'_n)\} : \alpha \to \beta &\text{ iff } \forall i.\ (d_i : \alpha \Rightarrow d'_i : \beta) \\ d : \tau_1 \wedge \tau_2 &\text{ iff } d : \tau_1 \text{ and } d : \tau_2 \\ d : \neg\tau &\text{ iff not } d : \tau \end{aligned}$$

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

It can be proved that $[\![\_]\!]_\mathcal{B}$ is a set-theoretic and structural interpretation; thus, it is a well-founded model.

4 Semantic subtyping

The bootstrap model $[\![\_]\!]_\mathcal{B}$ 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 shall 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. Now, the typing judgments for the language allow us to define a new natural set-theoretic interpretation of types, the one based on values $[\![\tau]\!]_\mathcal{V} = \{v \in \mathcal{V} \mid \vdash_\mathcal{B} v : \tau\}$ and then define a new ("real") subtyping relation:

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

The new relation $\leq_\mathcal{V}$ might in principle be different from $\leq_\mathcal{B}$. However, if the definitions of the model, of the language and of the typing rules have been carefully chosen, then the two subtyping relations coincide (see Theorem 1 at

the end of this section). 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 $\leq$.

An added value of this approach is the existence of an algorithm which decides the subtyping relation. The correctness of the semantic approach does not depend on the decidability, but it is still a nice property to have if we want to use our types in practice.

4.1 Typing Terms

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 keep into account the inheritance relation specified in the class definitions contained in $\widetilde{L}$. For notational convenience, we write "$a \in C$" to mean that there is a field declaration for 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.

In Table 1, we define the partial function $type(C)$ inductively 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. Let us notice that 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 because of the presence of **this**. Let us consider the following example (where, as usual, **int** $\leq$ **real** $\leq$ **compl**):

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** $x \times i$}
  real G() {**return this**.F()}
}

At run time, the function G returns a complex number, instead of a real. The point is that, when the method m is overloaded, we have to be sure that the return type should be a subtype of the original type, otherwise, due to the dynamic instantiation of **this**, there may be type errors. 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 for our calculus, 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. The rule (*subsum*) permits to derive for an expression e of type α_1 also a type α_2, when 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 $Val_{\mathbb{B}} \in \mathcal{C}$ 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

- $type(Object) = [\,]$;
- $type(C) = \rho$, 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 $\rho(a) = \alpha''$.

    Moreover, we assume that all the fields defined in ρ' and not declared in C appear at the beginning of ρ, with 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 $type(\_)$

have a non empty intersection. The rule (*var*) derives that x has type α, under the premise that the variable x has type α in the typing environment Γ. Let us now concentrate on the rules (*field*) and (*m-inv*): the 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; the 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 get the singleton record needed by using the subsumption rule. 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 [15], we can extend the type of an object at will (thus, types α'_i and μ'_j can be taken from any finite set of field- and method-types); in this way, we describe not only object's fields and methods, but also the types that cannot be assigned to it (assuming that the latter ones do not lead to a contradiction, i.e. a type semantically equivalent to $\mathbf{0}$). This is a technical trick needed to ensure that every non-zero type has at least one value of that type, a desirable property for the semantic subtyping approach. Rule (*rnd*) states that $\mathbf{rnd}(\alpha)$ is of type α. Finally, rule (*m-decl*) checks when a method declaration is acceptable for a class

Typing Expressions :

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

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

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

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

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 :

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

Typing Programs :

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

Table 2. Typing Rules

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.

Similarly to [15,10], the type checking relation is decidable.

4.2 Types as sets of values

Having defined the typing system, we can now interpret types as sets of values. So, 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.

Values in our calculus are constants or objects initialized by only passing values to their constructor. More formally:

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

However, this is not enough to assign values to every type of $\mathcal{T}$. In particular, a record type ρ will be interpreted as the set of objects that can be assigned type ρ. Objects are instantiations of classes in the class hierarchy $\widetilde{L}$ declared by the programmer. It is possible to assign to these objects the type of the class they instantiate. But, as the classes in $\widetilde{L}$ are finite, this means that we are able to interpret (inhabit with objects) just a finite number of record types. We aim at inhabiting all record types that can be inhabited with objects. Moreover, since we have not higher-order values, also the μ-types would not be inhabited by any value of the calculus. These are the main technical differences w.r.t. [15].

To overcome these problems, we define *pseudo-values* that are only used for interpreting types and deciding the subtyping relation. Pseudo-values essentially represent, in a different way, the λ-abstractions of [15]; like the values in the λ-calculus, our pseudo-values should be well-typed, closed, normal forms. So, before introducing pseudo-values, let us first define *normal forms*. These are (open) expressions that cannot reduce further and are produced by the following grammar:

$$w ::= x \mid c \mid \mathbf{new}\ \rho(\widetilde{w}) \mid w.a \mid w.m(w)$$

First of all, notice that every value $\mathbf{new}\ C(\widetilde{u})$ can be rewritten as $\mathbf{new}\ \mathit{type}(C)(\widetilde{u'})$, where $\widetilde{u'}$ is obtained from $\widetilde{u}$ by applying the same rewriting procedure. Using a record type ρ instead of a class C in the expression $\mathbf{new}\ \rho(\widetilde{u})$ permits us to inhabit all record types that can be inhabited.[3] It is important to 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 the record we are dealing with. For example, $\mathbf{new}\ [a : \mathbf{0}](u)$ 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 define pseudo-values, we are going to consider only closed, well-typed, normal forms. For α types, closed well-typed normal forms are the normal forms typed in the empty type environment. For μ types, we consider only the well-typed normal forms w that we "close" by assigning a type α to the free variable in w (indeed, since methods are unary and w represents the body of the method, a single type assignement is enough). Thus, the set of pseudo-values of type α is defined as

$$\mathcal{V}_\alpha = \{w \mid \vdash w : \alpha\}$$

whereas the set $\mathcal{V}_\mu$ of pseudo-values of type μ is defined as

$$\mathcal{V}_\mu = \{(\alpha, z) \mid z = \bot \vee z = w\}$$

where $\bot$ denotes non-termination and w is a well-typed normal form. Intuitively, pseudo-values of type μ represent methods that, taking an argument of type α, either non-terminate or return the normal form w, which can be assigned the return type of the method (see later on for a formalization of this idea). Notice

[3] Another solution to this problem is by assuming, for any record type ρ, the existence of a class C_ρ in the class hierarchy $\widetilde{L}$ such that $\mathit{type}(C_\rho) = \rho$. However, this solution would lead us to an infinite set of class declarations and this is not satisfying.

that this mix of types and terms exactly corresponds to the typed λ-abstraction of $\mathbb{C}$Duce, where both the type of the argument and the body of the function are specified. However, differently from $\mathbb{C}$Duce, we have no pseudo-value for inhabiting μ-types of the form $\alpha \to \alpha'$, for $\alpha' \simeq \mathbf{0}$, associated to non-terminating methods; this was the reason for introducing $\bot$.

Putting all together, the interpretation of a type τ is denoted as $[\![\tau]\!]_{\mathcal{V}}$ and is defined as follows:

$$[\![\alpha]\!]_{\mathcal{V}} = \{w \mid \vdash w : \alpha\}$$

$$[\![\mu]\!]_{\mathcal{V}} = \begin{cases} \varnothing & \text{if } \mu \simeq \mathbf{0} \\ \{(\alpha, z) \mid \forall i.\, \alpha \geq \alpha_i \ \wedge [(z = \bot \wedge \forall j.\, \alpha \not\geq \alpha'_j) \vee \\ \qquad\qquad (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha \vdash w : \beta_i)]\} \\ \quad \text{if } \mu = \bigwedge_{i=1\cdots n} \alpha_i \to \beta_i \wedge \bigwedge_{j=1\cdots m} \neg\left(\alpha'_j \to \beta'_j\right) \not\simeq \mathbf{0} \end{cases}$$

The interpretation of α-types follows the intuition that a type represents the set of its values. For μ-types, we essentially 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 once the input argument of the method is assigned type α (this is the meaning of the second disjunct in the second case of the definition for $[\![\mu]\!]_{\mathcal{V}}$). Since any μ-type can be assigned to non-terminating methods, it is natural to inhabit every μ-type with $\bot$. However, this fact could lead to inhabiting the empty type; this is prevented by the condition "$\forall j.\, \alpha \not\geq \alpha'_j$". For example, it forbids that any pair belonging to $(\alpha \to \beta)\wedge\neg(\alpha' \to \beta')$, also belongs to $(\alpha' \to \beta')\wedge\neg(\alpha \to \beta)$. Finally the condition "$\forall i.\, \alpha \geq \alpha_i$" in the definition of $[\![\_]\!]_{\mathcal{V}}$ ensures contra-variance of arrow types, as we see in the following example. Consider the class hierarchy:

class *Person* **extends** *Object* {**int** *age* ...}

class *Student* **extends** *Person* {**long** *matriculation_nr* ...}

class *Working_Student* **extends** *Student* {**string** *contract_nr* ...}

Consider now the arrow types *Student* $\to$ **long** and *Working_Student* $\to$ **long**. For the contra-variance of arrow types, we have that

$$\textit{Student} \to \textbf{long} \leq \textit{Working\_Student} \to \textbf{long}.$$

Then, it is easy to check that

$$[\![\textit{Student} \to \textbf{long}]\!]_{\mathcal{V}} \subseteq [\![\textit{Working\_Student} \to \textbf{long}]\!]_{\mathcal{V}}.$$

Indeed, $[\![\textit{Student} \to \textbf{long}]\!]_{\mathcal{V}}$ contains $(\textit{Object}, \bot)$, $(\textit{Student}, \bot)$ and $(\textit{Student}, x.\textit{matriculation\_nr})$ while $[\![\textit{Working\_Student} \to \textbf{long}]\!]_{\mathcal{V}}$ contains $(\textit{Object}, \bot)$, $(\textit{Student}, \bot)$, $(\textit{Working\_Student}, \bot)$, $(\textit{Student}, x.\textit{matriculation\_nr})$ and $(\textit{Working\_Student}, x.\textit{matriculation\_nr})$.

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

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

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

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

Table 3. Operational semantics

4.3 Closing the circle

As long as the subtyping relation is concerned, we have introduced the bootstrap model, that induces $\leq_{\mathcal{B}}$, and the interpretation of types as sets of values, that induces $\leq_{\mathcal{V}}$. The key result of our approach is that these two subtyping relations coincide and the proof can be found in [13].

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

5 Operational Semantics and Soundness of the Type System

The operational semantics is defined by the axioms and inference rules of Table 3, that are essentially the same as in [20]. The only notable differences are: (1) we use function *type* to extract the fields of an object, instead of defining an ad hoc function; (2) 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 $\textit{type}(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ as we are interested to have all the fields of the object instantiating class C and function $\textit{type}(C)$ 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

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

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. We choose the first way because, in our view, is more intuitive (even if less efficient); a more traditional modeling of overloading is possible and easy to model. Moreover, as already noted in Section 2.1, we use a simplified form of multimethods, where at most one declaration for every method name is present in every class. This simplifies the definition of function *body* given above and of function *type* given in Table 1. 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 [2]. A better alternative is the encoding of the more general setting of Section 6.2.

To complete the definition of the operational semantics, we need the straightforward rule *(r-ax)* for **rnd** and 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.

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. We proceed in the standard way, by stating the theorems of *subject reduction* and *progress*. Formal proofs are standard, and can be found in [13].

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

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

6 Discussion on the calculus

6.1 Recursive class definitions

It is possible to write recursive class definitions by assuming a special basic value **null** and a corresponding basic type **void**, having **null** as its only value. In Java, it is assumed that **void** is a sub-type of 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:

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

where α is the solution of the recursive type equation $\alpha = [val : \mathbf{int}, succ : (\alpha \vee \mathbf{void})]$. Now, we can create the list $\langle 1, 2 \rangle$ in the usual way, i.e. by writing the value **new** *intList*$(1, \mathbf{new}\ intList(2, \mathbf{null}))$.

6.2 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. However, we can encode the general scenario by adding one auxiliary subclass for every method definition. For instance, suppose that we want to define twice a multimethod m within the class A:

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

We then replace it with following declaration:

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

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. Thus, we can freely assume this encoding in the following sections.

6.3 Implementing Typical Java-like Constructs

We now want to 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. What we are going to present should show how naturally our framework can be used to implement these aspects on top of our core language.

The expression **if** e **then** e_1 **else** e_2 can be implemented by adding to the program the class definition:

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

where $\{\mathbf{true}\}$ and $\{\mathbf{false}\}$ are the singleton types containing only value **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

$$(\mathbf{new}\ \mathit{Test}()).m(e)$$

Notice that this term typechecks, since *test* has type $[m : (\{\mathbf{true}\} \to \alpha) \wedge (\{\mathbf{false}\} \to \alpha)] \simeq [m : (\{\mathbf{true}\} \vee \{\mathbf{false}\}) \to \alpha] \simeq [m : \mathbf{bool} \to \alpha]$. Indeed, in [15] it is proved that $(\alpha_1 \to \alpha) \wedge (\alpha_2 \to \alpha) \simeq (\alpha_1 \vee \alpha_2) \to \alpha$ and, trivially, $\{\mathbf{true}\} \vee \{\mathbf{false}\} \simeq \mathbf{bool}$.

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*:

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

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

$$(\mathbf{new}\ \mathit{InstOf}()).m_\alpha(e)$$

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

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

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

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

once we have added to the class the method

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

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

6.4 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

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

instead of $L_{intList}$ in Section 6.1. 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 to $L'_{intList}$ is $\mathit{intList} = [\mathit{val} : \textbf{int}, \mathit{succ} : (\mathit{intList} \vee \textbf{void})]$; 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 6.1.

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 to our semantic framework. To do that, to each class we add 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

$$X_C = \{s_1 s_2 \in CN^* : s_1 = \mathit{Object}\, C_1 \ldots C_k \,\&\, s_2 \in (CN \backslash \{\mathit{Object}, C_1, \ldots, C_k\})^*\}$$

where $\mathit{Object}, C_1, \ldots, C_k$ is the sequence of classes from *Object* to C in the class hierarchy, for $k \geq 0$ with $C_k = C$. 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:

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

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

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

$$X_C \;\; \mathbf{name}$$

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.

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, 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

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

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

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

7 Conclusion 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; by following [15], this can be done by deciding the problem of emptiness for *disjunctive normal forms* for types. Such forms have been defined in the full version of this paper [13] and algorithms similar to those in [15] can be adopted. This would be an intermediate step 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 ℂDuce. But other choices are possible too. The fact that we have assumed nondeterministic methods can also help in modeling a mutable state: as we have said, the input-output behaviour 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 [25]; since the framework of [15] is designed for a higher-order language, the theoretical machinery developed in loc.cit. should be easily adapted to the new fomalism.

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. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
8. G. Castagna. *Object-oriented programming: a unified foundation.* Progress in Theoretical Compuetr Science series, Birkäuser, Boston 1997.
9. G. Castagna. Semantic subtyping: Challenges, perspectives, and open problems. In *Proc. of ICTCS*, pages 1–20, 2005.
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. G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In *Proc of PPDP*, pages 198–199. ACM, 2005.
12. F. M. Damm. Subtyping with union types, intersection types and recursive types. In *Proc. of TACS*, pages 687–706. Springer, 1994.
13. 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`.
14. 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.
15. 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.
16. J. Gil and I. Maman. Whiteoak: introducing structural typing into Java. In *Proc. of OOPSLA*, pages 73–90. ACM, 2008.

17. H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *SIGPLAN Notices*, 36(3):67–80, 2001.
18. H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
19. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.
20. 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.
21. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.11*, 2008.
22. D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proc. of ECOOP*, pages 260–284. Springer, 2008.
23. D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In *Proc. of ESOP*, pages 95–111. Springer, 2009.
24. R. Milner. *Communicating and mobile systems: the $\pi-$calculus*. Cambridge University Press, New York, NY, USA, 1999.
25. M. Odersky, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, 2004.
26. K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.
27. D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. of POPL*, pages 40–53. ACM, 1997.
28. D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2003.