Semantic Subtyping for Objects and Classes

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

[1]Dipartimento di Informatica, Sapienza Università di Roma (Italy)

[2]PPS - Université Paris Diderot & CNRS (France)

ornela.dardha@gmail.com, gorla@di.uniroma1.it,
Daniele.Varacca@pps.jussieu.fr

ABSTRACT

There are two approaches for defining subtyping relations: the *syntactic* and the *semantic* one. In the semantic approach one starts from a model of the language of interest and an interpretation of types as subsets of the model. The subtyping relation is then defined as inclusion of sets denoting types. An orthogonal issue, typical of object-oriented languages, is the issue of *nominal* vs. *structural* subtyping. We aim to integrate structural subtyping with boolean connectives and semantic subtyping for a object-oriented core language and define a Java-like programming platform that exploits the benefits of both approaches, expressible in terms of code reuse and of compactness of program writing.

A *type system* for a programming language is a set of deduction rules that allows to derive a type for the terms of the language. Usually, type systems are based on a subtyping relation on types. There are two approaches for defining the subtyping relation: the *syntactic* approach and the *semantic* one. The syntactic approach is more common and consists in defining the subtyping relation 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. On the other hand, 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.

The first use of the semantic approach goes back to 15 years ago, as in [1, 2]. Hosoya and Pierce have also adopted this approach in [3, 4, 5] to define XDuce, an XML-oriented language designed specifically to transform XML documents in 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 [6, 7, 8] extend the XDuce language with first-class functions and arrow types and define and implement 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 $\pi-$calculus [9, 10]. Castagna, Varacca and De Nicola in [11] have used this technique to define the $\mathbb{C}\pi$ language, a variant of the asynchronous $\pi-$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* [12], that is the functional fragment of Java. Following the line of [8], we have defined from scratch the subtyping relation semantically, giving many important theoretical results. To understand the advantages of the usage of boolean connectives in an object-oriented language, let us consider the following example: suppose we are working with polygons as triangles, rectangles, rumbles etc. We want to define a method that takes a polygon and returns the length of its longest diagonal; of course this can be done only if the polygon has at least four sides. In Java this problem can be implemented in different ways, for example:

```
class Polygon {···}

class Triangle extends Polygon {···}

class Other_Polygons extends Polygon {
                              ⋮
   real diagonal(Other_Polygons p) {...}
                              ⋮
}

class Rectangle extends Other_Polygons {···}

class Rumble extends Other_Polygons {···}
                              ⋮
```

Another way to implement the problem is by means of an interface:

```
public interface Diagonal {
   real diagonal(Polygon p);
}

class Polygon {···}

class Triangle extends Polygon {···}

class Rectangle extends Polygon  implements Diagonal {···}

class Rumble extends Polygon  implements Diagonal {···}
                              ⋮
```

Let us now suppose that the class hierarchy has *Polygon* as the parent class and all other geometric figure classes that extend *Polygon*. Let us assume that this hierarchy is given and that it is not possible to change it afterwards. In this scenario it is more difficult to define such a method. The only way to do it in Java is by defining the method *diagonal* in the class *Polygon* and using an **instanceof**, for example inside a **try-catch** that throws an exception at run time if the argument passed to the method is a triangle.

With the use of boolean types, instead, the solution to the problem is more elegant: it is enough to define a method that has an argument of type "*Polygon* AND NOT *Triangle*" ($Polygon \wedge \neg Triangle$) that permits the typechecker to control at compile time the restrictions on types. A way to implement the problem using the boolean connectives is as follows:

class *Polygon* **extends** *Object* $\{\cdots\}$

class *Triangle* **extends** *Polygon* $\{\cdots\}$

class *Rectangle* **extends** *Polygon* $\{\cdots\}$

class *Rumble* **extends** *Polygon* $\{\cdots\}$

$\vdots$

```
class Diagonal extends Object {
   real diagonal(Polygon ∧ ¬Triangle p){...}
}
```

At this point, it is enough to invoke the method *diagonal* on an argument of type *Polygon*: if the polygon is not a *Triangle*, then the method returns the length of the longest diagonal; otherwise, if the polygon is a triangle, then there will be a type error at compile time.

As 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. Recently, a new approach has emerged, the one based on structural subtyping [13, 14, 15]. 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 the types in B. The definition of structural subtyping as inclusion of sets corresponding to fields and methods of a class brings us back to the definition of semantic subtyping, to which it fits perfectly. Moreover, with a minor effort, it is also possible to include in the framework the choice of using nominal subtyping without affecting the theory underlying our framework.

The aim of our work is to integrate structural subtyping with boolean connectives and semantic subtyping and define a Java-like programming platform that exploits the benefits of both approaches, expressible in terms of code reuse and of compactness of program writing. From the theoretical side, this calls for the definition of a set-theoretic model for the types and the study of the subtyping relation upon this model. So, algorithms for deciding the subtyping relation are needed and have to be explicitly developed. From the practical side, this calls for the development of a prototype programming environment where writing and evaluating the performances of code written in the new formalism.

References

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41, New York, NY, USA, 1993. ACM.

[2] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 687–706, London, UK, 1994. Springer-Verlag.

[3] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for xml. *SIGPLAN Not.*, 36(3):67–80, 2001.

[4] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[5] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

[6] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–199, New York, NY, USA, 2005. ACM.

[7] Giuseppe Castagna. Semantic subtyping: Challenges, perspectives, and open problems. In *ICTCS*, pages 1–20, 2005.

[8] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):1–64, 2008.

[9] Robin Milner. *Communicating and mobile systems: the $\pi-$calculus*. Cambridge University Press, New York, NY, USA, 1999.

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

[11] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the pi-calculus. *Theor. Comput. Sci.*, 398(1-3):217–242, 2008.

[12] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[13] Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 73–90, New York, NY, USA, 2008. ACM.

[14] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 260–284, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? an empirical study. In *ESOP*, pages 95–111, 2009.