Semantic Subtyping for Objects and Classes

Ornela Dardha, Daniele Gorla and Daniele Varacca

SAPIENZA UNIVERSITÀ DI ROMA

Introduction

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.

The Calculus

Types:

$$\alpha ::= \mathbf{0} \mid \mathbb{B} \mid [\widetilde{l:\tau}] \mid \alpha \wedge \alpha \mid \neg\alpha$$
$$\mu ::= \alpha \rightarrow \alpha \mid \mu \wedge \mu \mid \neg\mu$$
$$\tau ::= \alpha \mid \mu$$

Terms:

$$\begin{array}{lcl} L & ::= & \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\widetilde{\alpha\ a};\ K\ \widetilde{M}\} \\ K & ::= & C(\widetilde{\beta\ b}; \widetilde{\alpha\ a})\{\mathbf{super}(\widetilde{b}); \mathbf{this}.\widetilde{a} = \widetilde{a};\} \\ M & ::= & \alpha\ m(\alpha\ a)\{\mathbf{return}\ e;\} \\ e & ::= & x \mid c \mid e.a \mid e.m(e) \mid \mathbf{new}\ C(\widetilde{e}) \end{array}$$

Semantic Subtyping

- **Step 1:** *type constructors* are augmented with **0**, **1** and the *boolean connectives* $\vee$, $\wedge$ e $\neg$. Let $\mathcal{T}$ be the types algebra.
- **Step 2:** give a *set-theoretic model* of the type algebra: define an *interpretation function* $[\![]\!]_{\mathcal{B}} : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{B})$ for some set $\mathcal{B}$. The function $[\![]\!]_{\mathcal{B}}$ must satisfy:

$$[\![\tau_1 \vee \tau_2]\!]_{\mathcal{B}} = [\![\tau_1]\!]_{\mathcal{B}} \cup [\![\tau_2]\!]_{\mathcal{B}}$$
$$[\![\tau_1 \wedge \tau_2]\!]_{\mathcal{B}} = [\![\tau_1]\!]_{\mathcal{B}} \cap [\![\tau_2]\!]_{\mathcal{B}}$$
$$[\![\neg\tau]\!]_{\mathcal{B}} = \mathcal{B} \setminus [\![\tau]\!]_{\mathcal{B}}$$

subtyping induced by $\mathcal{B}$:

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

- **Step 3:** find an algorithm that decides the subtyping relation.
- **Step 4:** consider the subtyping relation and the typing rules and deduce typing judgments $\Gamma \vdash_{\mathcal{B}} e : \tau$ for the language.
- **Step 5:** typing judgments allow us to define a new natural interpretation, types as set of values:

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

subtyping induced by $\mathcal{V}$:

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

Closing the circle

$$\tau_1 \leq_{\mathcal{B}} \tau_2 \overset{proved}{\iff} \tau_1 \leq_{\mathcal{V}} \tau_2$$

Advantages of boolean connectives in object-oriented languages

Example: suppose we are working with *polygons: triangles, squares, rumbles* etc. We want to define a method *diagonal* that given a *polygon* computes its longest diagonal. Of course, this is possible only if the *polygon* is **not** a *triangle*. In Java this can be done in different ways, for example:

class *Polygon* {· · ·}

class *Triangle* **extends** *Polygon* {· · ·}

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

Using interfaces:

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

class *Polygon* {· · ·}

class *Triangle* **extends** *Polygon* {· · ·}

class *Square* **extends** *Polygon* **implements** *Diagonal* {· · ·}

class *Rumble* **extends** *Polygon* **implements** *Diagonal* {· · ·}

⋮

Things are easier when done *ad-hoc*. **But it is not always this way...** Suppose the class-hierarchy is already declared and it is not possible to modify it afterwards. The situation is as follows:

class *Polygon* **extends** *Object* {· · ·}

class *Triangle* **extends** *Polygon* {· · ·}

class *Square* **extends** *Polygon* {· · ·}

class *Rumble* **extends** *Polygon* {· · ·}

⋮

It is more complicated to define the method *diagonal*. One can define this method in the class *Polygon* and use an **instance-of** construct and handle **exceptions**. If a *triangle* is passed to the method, then an exception is thrown and will be handled at run-time. Or...

Let's use connectives!!

Define a method with argument type $Polygon \wedge \neg Triangle$.

class *Diagonal* **extends** *Object* {
real *diagonal*($Polygon \wedge \neg Triangle$ *p*){. . .}
}

If a *polygon* **not** *triangle* is passed to the method *diagonal*, then the longest diagonal is computed; otherwise, if a *triangle* is passed to the method, then a type-error at compile time will occur.

Results and Conclusions

- We considered the functional fragment of Java.
- We gave a set-theoretic interpretation in $\mathcal{B}$ that induces $\leq_{\mathcal{B}}$.
- Next, a new interpretation in $\mathcal{V}$ is given, (types as set of values) that induces $\leq_{\mathcal{V}}$.
- **Theorem:** $\leq_{\mathcal{B}} \iff \leq_{\mathcal{V}}$

Nominal vs. Structural

There are two approaches to subtyping specific to object-oriented languages:

- Nominal: *A* is a subtype of *B* if and only if it is declared to be so (*declarative*).
- Structural: *A* is a subtype of *B* if and only if its fields and methods are a **superset** of the fields and methods of *B* and their types are subtypes of types in *B* (*intrinsic*).

Observation: it is natural to integrate *structural subtyping* with boolean connectives and *semantic subtyping*, exploiting their benefits.

Future work

- We aim at constructing universal models of types.
- Prove properties of $[\![]\!]_{\mathcal{V}}$.
- Implement a prototype OO language $\mathbb{J}$Duce that uses boolean connectives and a semantically defined subtyping.