An Approach to Behavioral Subtyping Based on Static Analysis

Francesco Logozzo [1]

STIX - École Polytechnique
F-91128 Palaiseau, France

Abstract

In mainstream object oriented languages the subclass relation is defined in terms of subtyping, i.e. a class A is a subclass of B if the type of A is a subtype of B. In this paper this notion is extended to consider arbitrary class properties obtained by a modular static analysis of the class. In such a setting, the subclass relation boils down to the order relation on the abstract domain used for the analysis of the classes. Furthermore we show how this approach yields a more semantic characterization of class hierarchies and how it can be used for an effective modular analysis of polymorphic code.

Keywords: Abstract Interpretation, Inheritance, Object Oriented Languages, Semantics, Specialized Application Languages, Static Analysis

1   Introduction

Subclassing is one of the main features of object oriented languages. It allows a form of incremental programming and of code reuse. Moreover it allows the structuring of the code in a hierarchy, so that the classes composing a program (or a library) are organized in a subclass hierarchy. The traditional definition of the subclass relation is that a class A is subclass of class B if its type is a subtype of that of B. Stated otherwise this means that an object that belongs to A can be used in any context that requires an object of B without causing a type-error at runtime. However, the subtyping relation is not strong enough to ensure, for instance, that an object of A does not cause a division by zero,

---

[1]   Email: Francesco.Logozzo@Polytechnique.fr

if the B's object did not. Behavioral subtyping tries to overcome this problem [7,10].

Roughly speaking the behavioral subtype relationship guarantees that no unexpected behavior occurs when subtype objects replace supertype's ones. The essential idea is to annotate the class source code with a property in a suitable formal language. Such a property is called the behavior type of the class [7]. Then the behavioral subtyping relation is defined in terms of property implication: A is a subclass of B if its behavioral type implies that of B. The checking of this implication can be done in several ways: by a hand-proof [7], a theorem prover [6] or even at runtime [10]. However, most of the times the formal correspondence between the class semantics and the hand-provided behavioral type is neglected.

In this paper we present an approach to behavioral subtyping based on a modular static analysis [8,9]. The main idea is to analyze a class on a suitable abstract domain to infer a class invariant as well as methods preconditions and postconditions. We call the result of the analysis of A the observable of A, $\mathcal{O}(A)$. An observable is a sound approximation of the class semantics, thus it is a behavioral type of A. The correspondence between the semantics of A and $\mathcal{O}(A)$ is straightforwardly given by the soundness of the static analysis. The behavioral subtyping relation boils down to the order $\bar{\sqsubseteq}$ on the abstract domain: given two classes A and B, then $\mathcal{O}(A) \bar{\sqsubseteq} \mathcal{O}(B)$ means that A preserves the behavior of B. In other words A is a behavioral subtype of B.

Our approach to behavioral subtyping has several advantages. At first as it is based on static analysis it does not require any human intervention for the annotation of the source code. Furthermore, the observable is ensured to be a sound approximation of the class semantics and it saves programmer time. Eventually, as the order relation $\bar{\sqsubseteq}$ is decidable it can be automatically checked. Thus there is no need of using a theorem prover or to rely on unsound methods as runtime assertion monitoring [10]. Furthermore the definition of the behavioral subtyping in the abstract interpretation framework allows to use standard techniques as for instance domain refinement [4] in order to systematically improve the precision of the observables.

1.1 *Motivating Examples*

As an example, let us consider the classes in Fig. 1. They implement different kinds of bags. They have a method to add an element to the container, add(e) and addSq(e), and to extract an element from it, remove(). However, they differ in the handling of the elements: the method remove() of Queue returns the elements in the same order they have been inserted whereas that of Stack, PosStack and SqStack returns them in the reverse order. Moreover PosStack

```
class Stack is                      class Queue is

s : list of int;                    s : list of int;

init() : s = []                     init() : s = [];

add(e) : s = e :: s                 add(e) : s = s :: e

remove() : let s = e :: ls in       remove() : let s = e :: ls in

    s = ls;                             s = ls;

    return e                            return e

        (a) Stack                           (b) Queue
```

```
                                    class SqStack is

class PosStack is                   s : list of int;

s : list of int;                    init() : s = []

init() : s = []                     add(e) : s = |e| :: s

add(e) : s = |e| :: s               addSq(e) : s = (e * e) :: s

remove() : let s = e :: ls in       remove() : let s = e :: ls in

    s = ls;                             s = ls;

    return e                            return e

        (c) PosStack                        (d) SqStack
```

Fig. 1. The paper running examples

and `SqStack` contain only positive integers and `SqStack` has a further method that inserts the square of its argument. For the sake of simplicity we do not consider such errors, as removing an element from an empty container.

1.1.1 Class Hierarchy.

It is evident that the four classes have different behaviors. However the three are not totally unrelated, so which is the relation between them? Which are the admissible class hierarchies? To put it another way, when is it safe to replace an object `s` of `Stack` with an object `q` of `Queue`? The answer depends on the meaning of "safe". In type theory "safe" means that the use of `q` at the

place of s will not cause a run-time type error, if s did not cause one. Thus in the example, the classes Stack, PosStack and Queue have the same type so that for instance Stack may be a subtype of Queue and conversely. Only SqStack has to be a subtype of Stack, PosStack or Queue, due to method addSq. This is the only constraint on the possible class hierarchies.

On the other hand, if the context requires that the values extracted from the bag are in the reverse order with respect to the insertion one then it is not "safe" anymore to replace s with q. Thus the order of the elements of a bag is a property, different from types, that induces a different subclassing relationship. In particular, from this point of view Stack and SqStack exhibit the same property, so that Stack may be defined as a subclass of SqStack. Therefore, the admissible class hierarchies are different from that allowed by the subtyping relation.

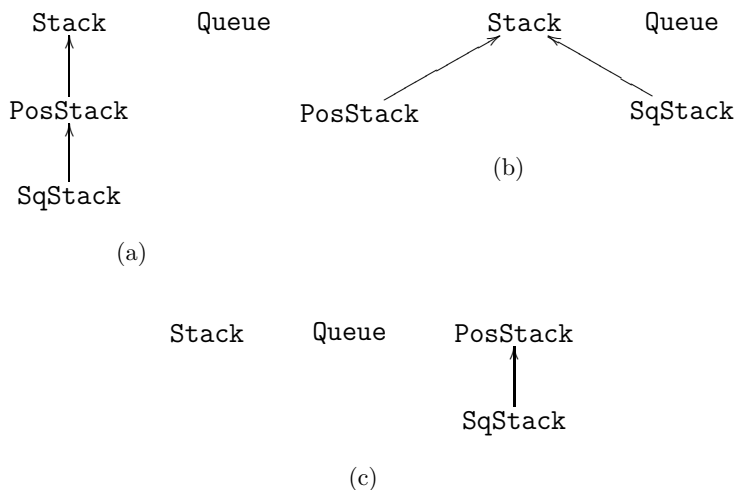### 1.1.2   *Systematic Refinement of the Class Hierarchy.*

Types and element ordering are both properties of classes that can be discovered once they are analyzed on suitable abstract domains, say $\bar{\mathsf{T}}$ and $\bar{\mathsf{S}}$ respectively. The two domains can be combined together using the reduced product $\bar{\mathsf{P}} = \bar{\mathsf{T}} \times \bar{\mathsf{S}}$ [4]. Thus, using the more precise abstract domain $\bar{\mathsf{P}}$ it is possible to infer more precise class properties. In the example SqStack can only be a subclass of Stack or of PosStack. However it is still admissible to have Stack subclass of PosStack and *vice versa*. This is essentially a consequence of the fact that $\bar{\mathsf{P}}$ does not capture the sign of the elements in s. Therefore, $\bar{\mathsf{P}}$ can be combined with the $\bar{\mathsf{Sign}}$ abstract domain [3] in order to capture such a property: $\bar{\mathsf{R}} = \bar{\mathsf{P}} \times \bar{\mathsf{Sign}}$. Thereafter, using $\bar{\mathsf{R}}$ we obtain that Stack can never be subclass of PosStack as it does not preserve the property that all the elements in s are positive integers. Only four class hierarchies preserve the properties captured by $\bar{\mathsf{R}}$: the trivial one in which the subclass relation is the identity and the three listed in Fig. 2.

1.1.3   *Modular Verification.*

The initial motivation of our work was the application of behavioral subtyping to the modular analysis of polymorphic object oriented code, robust with respect to the addition of subclasses. Consider for example the following function that references an object of type PosStack:

$$\texttt{sqrt(PosStack p)} : \texttt{return } \sqrt{\texttt{p.remove()}}.$$

One would like to prove sqrt correct for all possible future subclasses of PosStack, as all these may be passed as a parameter. This is possible if the subclasses do not violate the PosStack property that the elements are always

```
   Stack        Queue              Stack        Queue
     ↑                               ↖        ↗
  PosStack                     PosStack          SqStack
     ↑
                                         (b)
  SqStack

        (a)


        Stack      Queue      PosStack
                                 ↑
                              SqStack

                 (c)
```

Fig. 2. Admissible class hierarchies using $\bar{\mathsf{R}}$

positive. It is evident that the subtyping-based subclass relation is too weak to ensure this property.

However, if only subclass relations based on the properties encoded in $\bar{\mathsf{R}}$ are allowed, then all the subclasses of PosStack preserve the required invariants. This reduces the proof that sqrt never performs the square root of a negative number to proving it with PosStack as an argument.

2   Abstract Semantics

A class A can be seen as a triple $\langle \mathsf{F}, \mathtt{init}, \mathsf{M} \rangle$ where init is the class constructor, F is a set of fields, and M is a set of methods. For the sake of generality we assume that untyped fields and methods. The *concrete* semantics $\mathbb{C}[\![\mathsf{A}]\!]$ of a class A can be given as a set of trees. Roughly, each tree is the semantics of an instance of A and it represents all the possible interactions between the context and the object. For a formal definition of the class semantics we refer the reader to [8].

Given a class A, a static analysis of A is an approximation of its concrete semantics. In the abstract interpretation framework [3] this can be formalized by an abstract semantic function $\bar{\mathbb{C}}[\![\mathsf{A}]\!]$ defined on an abstract domain $\langle \bar{\mathsf{D}}, \sqsubseteq \rangle$. The elements of $\bar{\mathsf{D}}$ are computer-representable approximations of the concrete properties and the $\sqsubseteq$ order on the abstract domain $\bar{\mathsf{D}}$ is the abstract counterpart for the logical implication. The correspondence between the concrete and abstract domain is given by a Galois connection $\langle \alpha, \gamma \rangle$. The concretiza-

tion of the abstract semantics, $\gamma(\bar{\mathbb{C}}[\![A]\!])$, expresses the abstract information available about the semantics of A in concrete terms. It should be a sound approximation of the concrete semantics, i.e. $\mathbb{C}[\![A]\!] \subseteq \gamma(\bar{\mathbb{C}}[\![A]\!])$.

A behavioral type of a class A is a property of the semantics of A. In this paper we consider behavioral types that are the result of a static analysis of A, and we define them as *observables* of the class: $\mathcal{O}(A) = \bar{\mathbb{C}}[\![A]\!]$.

The advantage of defining the behavioral type of a class as the result of a static analysis of its source is that it can be automatically inferred. Moreover, given two classes A and B, it is sufficient to check if $\mathcal{O}(A) \bar{\sqsubseteq} \mathcal{O}(B)$ in order to verify if they are in the behavioral subtype relation. In the following we define a decidable $\bar{\sqsubseteq}$ relation.

2.1 Static Analysis of a Class

Given a class $A = \langle \mathtt{init}, F, M \rangle$, an abstract domain $\langle \bar{P}, \bar{\preceq} \rangle$ and an abstraction of the methods semantics $\bar{\mathrm{m}}[\![\cdot]\!] \in [M \to \bar{P} \to \bar{P}]$ it is possible to modularly analyze A in order to infer a class invariant as well as methods preconditions and postconditions [9,8]. In fact, let us consider the following equation system:

$$I = I_0 \bar{\sqcup} \bar{\bigsqcup_i} I_i$$
$$I_0 = \bar{\mathrm{m}}[\![\mathtt{init}]\!]$$
$$I_i = \bar{\mathrm{m}}[\![\mathtt{m_i}]\!](I) \qquad \mathtt{m_i} \in M.$$

It can be solved using standard fixpoint iteration techniques [3]. Then it is possible to show that I is a class invariant and $\{I_i\}$ are the methods postconditions [9]. The method preconditions are obtained by a backward analysis starting from the postcondition: $P_i = \bar{\mathrm{m}}^<[\![\mathtt{m_i}]\!](I_i)$. Finally, the result of the static analysis of A is:

$$\bar{\mathbb{C}}[\![A]\!] = \langle I, \{\mathtt{m_i} : P_i \to I_i \mid \mathtt{m_i} \in M\} \rangle.$$

Remark 2.1 It is possible to extend the equation system above in order to derive a sound class invariant even when an object exposes a part of its state to the context [9]. Roughly, the abstract domain $\bar{P}$ must be refined in order to capture escape information Nevertheless, in order to focus ourselves on the topic of the paper, we make the simplifying assumption that an object does not expose (a part of) its state.

2.2   Domain of Observables

The domain of the observables, $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq} \rangle$, is built on the top of the domain used for the analysis, $\langle \bar{\mathsf{P}}, \bar{\preceq} \rangle$. Hence the elements belong to the set

$$\bar{\mathsf{O}} = \{ \langle I, \{ \mathtt{m_i} : P_i \to I_i \} \rangle \mid I \in \bar{\mathsf{P}}, \forall i.\ P_i, I_i \in \bar{\mathsf{P}} \}.$$

The order relation $\bar{\sqsubseteq}$ is defined point-wise. Let $\mathsf{o_1} = \langle I, \{ \mathtt{m_i} : P_i \to I_i \} \rangle$ and $\mathsf{o_2} = \langle J, \{ \mathtt{m_j} : Q_j \to J_j \} \rangle$ be two elements [2] of $\bar{\mathsf{O}}$. Then

$$\mathsf{o_1} \bar{\sqsubseteq} \mathsf{o_2} \iff I \bar{\preceq} J \wedge (\forall \mathtt{m_i}.\ Q_i \bar{\preceq} P_i \wedge I_i \bar{\preceq} J_i).$$

Roughly speaking, if $\mathsf{o_1}$ and $\mathsf{o_2}$ are the observables of two classes $\mathtt{A}$ and $\mathtt{B}$ then the order $\bar{\sqsubseteq}$ ensures that $\mathtt{A}$ preserves the class invariant of $\mathtt{B}$ and that the methods of $\mathtt{A}$ are a "safe" replacement of those with the same name in $\mathtt{B}$. Intuitively, the precondition condition says that if the context satisfies Q_i then it satisfies the inherited method precondition P_i too. Thus the inherited method can be used in any context where its ancestor can. On the other hand, the postcondition of the inherited method may be stronger than that of the ancestor.

   Having defined $\bar{\sqsubseteq}$, it is routine to check that if $\bar{\mathsf{P}}$ is a complete lattice then $\langle \bar{\bot}, \{ \mathtt{m_i} : \bar{\top} \to \bar{\bot} \} \rangle$ is the smallest element of $\bar{\mathsf{O}}$ and $\langle \bar{\top}, \{ \mathtt{m_i} : \bar{\bot} \to \bar{\top} \} \rangle$ is the largest one. The join and the meet operations can be defined point-wise. Thus the abstract domain $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq} \rangle$ is a complete lattice.

   Moreover, let us suppose that the order relation $\bar{\preceq}$ is decidable. This is the case of an abstract domain used for an effective static analysis. As $\bar{\sqsubseteq}$ is defined in terms of $\bar{\preceq}$ and the universal quantification ranges on a finite number of methods then $\bar{\sqsubseteq}$ is decidable too. To sum up, we have shown:

Theorem 2.2 *Let $\langle \bar{\mathsf{P}}, \bar{\preceq} \rangle$ be a complete lattice. Then $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq} \rangle$ is a complete lattice. Moreover, if $\bar{\preceq}$ is decidable then $\bar{\sqsubseteq}$ is decidable.*
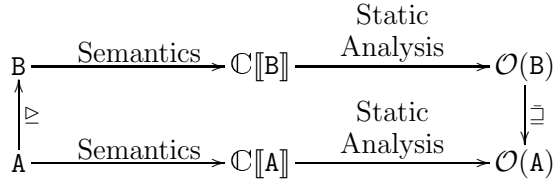
2.3   Subclassing through Observables

It is now possible to formally give the definition of subclassing as inclusion relation between elements of the abstract domain:

Definition 2.3 Let $\mathtt{A}$ and $\mathtt{B}$ be two classes, $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq} \rangle$ a domain of observables. Then $\mathtt{A}$ is a subclass of $\mathtt{B}$, $\mathtt{A} \unlhd \mathtt{B}$, with respect to the properties encoded by $\bar{\mathsf{O}}$ iff $\mathcal{O}(\mathtt{A}) \bar{\sqsubseteq} \mathcal{O}(\mathtt{B})$.

---

[2] We use the same index for methods with the same name. For instance P_i and Q_i are the preconditions for the homonym method $\mathtt{m_i}$ of $\mathsf{o_1}$ and $\mathsf{o_2}$.

Observe that when $\langle \bar{\mathsf{O}}, \bar{\sqsubseteq} \rangle$ is instantiated with the types abstract domain [2] then the relation defined above coincides with the traditional subtyping-based definition of subclassing [1].

The Def. 2.3 can be visualized by the following diagram:

$$
\begin{array}{ccccc}
\mathtt{B} & \xrightarrow{\text{Semantics}} & \mathbb{C}[\![\mathtt{B}]\!] & \xrightarrow[\text{Analysis}]{\text{Static}} & \mathcal{O}(\mathtt{B}) \\[2pt]
{\scriptstyle \unrhd}\big\uparrow & & & & \big\downarrow{\scriptstyle \sqsupseteq} \\[2pt]
\mathtt{A} & \xrightarrow{\text{Semantics}} & \mathbb{C}[\![\mathtt{A}]\!] & \xrightarrow[\text{Analysis}]{\text{Static}} & \mathcal{O}(\mathtt{A})
\end{array}
$$

This diagram essentially shows how the concept of subclassing is linked to the semantics of classes. It states that when the abstract semantics of A and B are compared, that of A implies the one of B. That means that A refines B w.r.t. the properties encoded by the abstract domain $\bar{\mathsf{O}}$. This is in accord with the mundane understanding of inheritance which states that a subclass is a specialization of the ancestor.

Moreover we have made no-hypothesis on the abstraction of the concrete semantics. In particular we do not differentiate between history properties and state properties, unlike [7], the two being just different abstractions of the concrete semantics. In fact, history properties correspond to trace abstractions and state properties to state abstractions.

2.3.1 Static Checking of Behavioral Subtyping.

The main advantage of our approach is that the subclassing relation can automatically be checked by a compiler: the derivation of class observables is automatic and their inclusion is decidable. As a consequence a compiler can accept subclasses only if they preserve the parent *behavior*. For instance, this is in the spirit of Eiffel subclassing mechanism [10]. However, the specification of Eiffel requires to check the preservation of the ancestor invariants at runtime. An interesting future work can be the extension of our work on subclassing to the Eiffel language.

2.3.2 Modular Verification.

A major advantage of having the compiler which rejects subclass definitions, that do not preserve the parent properties, is that it enables a form of modular analysis for polymorphic functions. Consider the following polymorphic function f, that references an object of type B:

$$\mathtt{f(B\ b)\ :\ \ldots\ b.m(v)\ldots.}$$

Now, suppose to analyze it on the $\langle \bar{\mathsf{P}}, \bar{\preceq} \rangle$ abstract domain. If the analysis is performed using B then the call $\mathsf{b.m(v)}$ resolves to the invocation of the method $\mathsf{m_B}$ of B. Having an observable of B, the precondition Q and the post-condition J of $\mathsf{m_B}$ can be used in the analysis, so that the body of the methods does not need to be analyzed again. Thus, if $\bar{\mathsf{v}} \in \bar{\mathsf{P}}$ is the approximation of the concrete values taken by the variable v then

$$\bar{\mathsf{v}} \bar{\preceq} Q \Longrightarrow \bar{\mathsf{m}}[\![\mathsf{m_B}]\!](\bar{\mathsf{v}}) \bar{\preceq} J.$$

The result of such an analysis is valid for all the invocations $\mathsf{f(a)}$ where a is an instance of a class $\mathsf{A} \trianglelefteq \mathsf{B}$. This can be shown as follows. If $\mathsf{A} \trianglelefteq \mathsf{B}$ then $\mathcal{O}(\mathsf{A}) \sqsubseteq \mathcal{O}(\mathsf{B})$. Then, by definition of the order relation $\sqsubseteq$ the method $\mathsf{m_A}$ of A is such that $\mathsf{m_A} : P \rightarrow I$ with $Q \bar{\preceq} P$ and $I \bar{\preceq} J$. So:

$$\bar{\mathsf{v}} \bar{\preceq} Q \bar{\preceq} P \Longrightarrow \bar{\mathsf{m}}[\![\mathsf{m_A}]\!](\bar{\mathsf{v}}) \bar{\preceq} I \wedge I \bar{\preceq} J.$$

Thus J is a sound approximation of the semantics of the method $\mathsf{m_A}$. As a matter of fact we have proved the following theorem:

Theorem 2.4 *Let* A *and* B *two classes such that* $\mathsf{A} \trianglelefteq \mathsf{B}$. *Let* $\mathsf{m_B}$ *be a method of* B *and* $\mathsf{m_A}$ *the homonym method that belongs to* A. *Let* $\mathsf{m_B} : Q \rightarrow J$ *and* $\mathsf{m_A} : P \rightarrow I$. *Then*

$$\forall \bar{\mathsf{v}} \in \bar{\mathsf{P}}. \ \bar{\mathsf{v}} \bar{\preceq} Q \Longrightarrow \bar{\mathsf{m}}[\![\mathsf{m_A}]\!](\bar{\mathsf{v}}) \bar{\preceq} J.$$

Hence the analysis of polymorphic code using the superclass is enough to state that the result is valid for all the subclasses. So, it is not necessary to reanalyze the code for each subclass of B.

2.3.3 Domain Refinement.

A further advantage of formalizing the behavioral subtyping in the abstraction interpretation framework is that it is possible to apply well-known abstract domain refinement techniques [4,5] in order to improve the precision of the observables. Hence having more fine-grain class hierarchies. In particular, the use of the reduced product is practical for refining the precision of the captured properties.

An abstract domain of observables must, at least, encapsulate the types abstract domain $\bar{\mathsf{T}}$. On the other hand we have argued before how a further abstract domain $\bar{\mathsf{D}}$ is needed to capture non-typing properties, e.g. the sign of the field values. Then the domain of observables can be built on the top of the reduced product of the two: $\bar{\mathsf{P}} = \bar{\mathsf{T}} \times \bar{\mathsf{D}}$. As a consequence, from a well-known result in abstract interpretation (cfr. Th. 10.1.0.2 of [4]) it follows that $\bar{\mathsf{P}}$ is a domain more precise than types, so that the resulting $\trianglelefteq$ relation is more precise than the subtyping one.

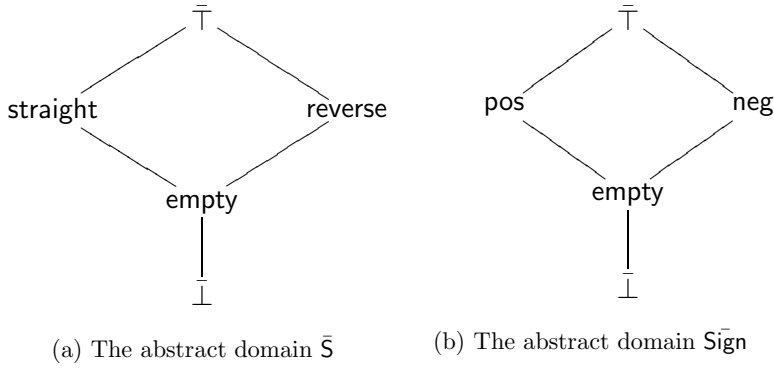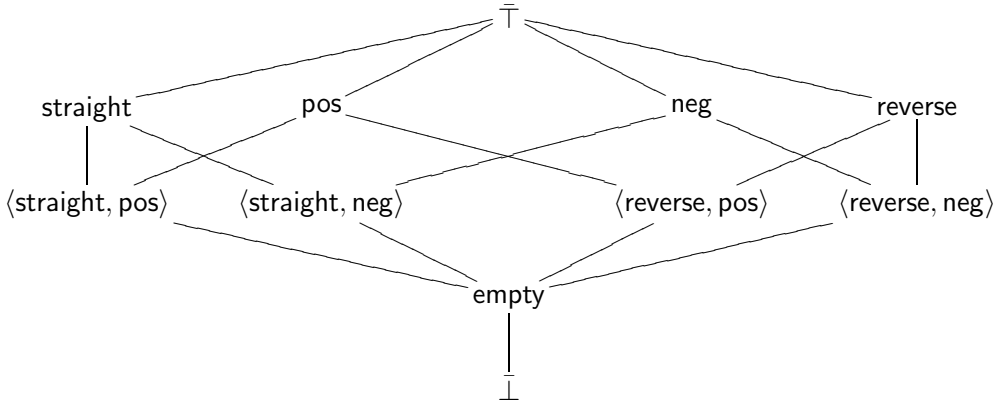(a) The abstract domain $\bar{\mathsf{S}}$ (b) The abstract domain $\bar{\mathsf{Sign}}$

Fig. 3. Abstracts domains expressing the order of elements and their sign

Fig. 4. The abstract domain $\bar{\mathsf{H}} = \bar{\mathsf{S}} \times \bar{\mathsf{Sign}}$

3   Application to the Examples

In this section we show how the definition of the previous section applies to the
examples of Sect. 1.1. At first we show how when instantiating the underline
abstract domain with types, the definition of $\trianglelefteq$ reduces to the traditional
subtype-based one.

It is known that types can be seen as an abstract interpretation [2]. We
call $\bar{\mathsf{T}}$ the corresponding abstract domain [3]. Then if we instantiate the Def.

---

[3] Actually in the cited paper the author considered several abstract domains corresponding
to several typing system. The abstract domain $\bar{\mathsf{T}}$ that we consider is that of Church/Curry
monotypes.

2.3 with the abstract domain $\bar{\mathsf{T}}$ we obtain that:

$$\mathcal{O}(\texttt{Stack}) = \mathcal{O}(\texttt{Queue}) = \mathcal{O}(\texttt{PosStack}) =$$
$$\{\langle \texttt{s} : \texttt{list of int}\rangle,$$
$$\{\texttt{init} : \texttt{void} \to \texttt{void}; \ \texttt{add} : \texttt{int} \to \texttt{void};$$
$$\texttt{remove} : \texttt{void} \to \texttt{int}\}\},$$
$$\mathcal{O}(\texttt{SqStack}) = \{\langle \texttt{s} : \texttt{list of int}\rangle, \ \{\texttt{init} : \texttt{void} \to \texttt{void};$$
$$\texttt{add}, \ \texttt{addSq} : \texttt{int} \to \texttt{void}; \ \texttt{remove} : \texttt{void} \to \texttt{int}\}\}$$

Therefore the only constraint on the definition of the subclassing relation is that SqStack cannot be the ancestor of any of the other three. This is because $\mathcal{O}(\texttt{SqStack})$ is a subtype of $\mathcal{O}(\texttt{Stack})$ [1].

A different subclassing relation can be obtained using the abstract domain in Fig. 3(a), whom intuitive meaning is to consider if the elements of the list are inserted at the head or tail position. It is worth noting that the order of the elements is a history property. In that case, the observables using $\bar{\mathsf{S}}$ are:

$$\mathcal{O}(\texttt{Stack}) = \mathcal{O}(\texttt{PosStack}) = \{\langle \texttt{s} : \mathsf{reverse}\rangle,$$
$$\{\texttt{init} : \bar{\bot} \to \mathsf{empty}; \ \texttt{add} : \mathsf{reverse} \to \mathsf{reverse};$$
$$\texttt{remove} : \mathsf{reverse} \to \mathsf{reverse}\}\},$$
$$\mathcal{O}(\texttt{SqStack}) = \{\langle \texttt{s} : \mathsf{reverse}\rangle, \{\texttt{init} : \bar{\bot} \to \mathsf{empty};$$
$$\texttt{add}, \ \texttt{addSq} : \mathsf{reverse} \to \mathsf{reverse};$$
$$\texttt{remove} : \mathsf{reverse} \to \mathsf{reverse}\}\},$$
$$\mathcal{O}(\texttt{Queue}) = \{\langle \texttt{s} : \mathsf{straight}\rangle, \{\texttt{init} : \bar{\bot} \to \mathsf{empty};$$
$$\texttt{add} : \mathsf{straight} \to \mathsf{straight};$$
$$\texttt{remove} : \mathsf{straight} \to \mathsf{straight}\}\}.$$

In this last example, it happens that for instance Queue and Stack can never be in the subclass relation as neither $\mathcal{O}(\texttt{Queue}) \ \bar{\sqsubseteq} \ \mathcal{O}(\texttt{Stack})$ nor $\mathcal{O}(\texttt{Stack}) \ \bar{\sqsubseteq} \ \mathcal{O}(\texttt{Queue})$. However nothing avoids to have Stack $\trianglelefteq$ PosStack as $\bar{\mathsf{S}}$ do not capture the sign of the elements in s, but just the order in which they are inserted. Therefore it is possible to refine $\bar{\mathsf{S}}$ using the domain $\bar{\mathsf{Sign}}$ of Fig. 3(b). In that case we consider the domain $\bar{\mathsf{H}}$ of Fig. 4 that is the reduced product of the two: $\bar{\mathsf{H}} = \bar{\mathsf{S}} \times \bar{\mathsf{Sign}}$. Thus the resulting observables for the two

classes are:

$$\mathcal{O}(\texttt{Stack}) = \{\texttt{s} : \textsf{reverse}, \{\texttt{init} : \bar{\bot} \to \textsf{empty};$$
$$\texttt{add} : \textsf{reverse} \to \textsf{reverse};$$
$$\texttt{remove} : \textsf{reverse} \to \textsf{reverse}\}\},$$
$$\mathcal{O}(\texttt{PosStack}) = \{\texttt{s} : \langle\textsf{reverse}, \textsf{pos}\rangle, \{\texttt{init} : \bar{\bot} \to \textsf{empty};$$
$$\texttt{add} : \langle\textsf{reverse}, \textsf{pos}\rangle \to \langle\textsf{reverse}, \textsf{pos}\rangle;$$
$$\texttt{remove} : \langle\textsf{reverse}, \textsf{pos}\rangle \to \langle\textsf{reverse}, \textsf{pos}\rangle\}\}.$$

It is routine to check that $\texttt{Stack} \not\preceq \texttt{PosStack}$. Eventually, the subclass relation that brings to the class hierarchies in Fig. 2 is obtained considering the properties encoded by the abstract domain $\bar{\mathsf{R}} = \bar{\mathsf{T}} \times \bar{\mathsf{H}} = \bar{\mathsf{T}} \times \bar{\mathsf{S}} \times \bar{\mathsf{Sign}}$.

4   Conclusions and Future Work

In this work we have presented an approach to the behavioral subtyping based on a modular static analysis of classes' source. In particular we have shown how the subclassing relation can be defined in terms of the order on the underlying abstract domain. Our approach has several advantages over traditional subtyping and behavioral subtyping: the class behavioral type is automatically inferred, the subtyping is decidable, it is more semantically characterized and it is formulated in the abstract interpretation framework so that well-known techniques on the composition and refinement of abstracts domain can be used. Moreover, in this setting the problem of analyzing, and hence verifying, polymorphic code is by far more simple. In fact as every subclass extension preserves, by definition, the ancestor invariant then it is not required to reanalyze the code once a new subclass is defined.

However, an open problem is the choice of the abstract domain used for the inference of observables. For what concerns general purposes object oriented languages, it is difficult to fix in advance the properties that one wishes to be preserved by subclasses. Types have been shown to be effective for that purpose, so that an abstract domain of observables must at least include them. One can think to continue in that direction considering other runtime errors, e.g. division by zero, overflow or null-pointer dereferencing, so that given a class, all its subclasses are assured not to introduce runtime errors.

On the other hand we are trustful that our approach can be effective for the design and the development of problem-specific object oriented languages. As an example, let us consider a language for smartcards programming. In this setting security is important, so that a wished property is that if a subclass does not reveal a secret, so do the subclasses. In that case, we can use a

domain of observables able to capture security and information-flow properties. Embedded systems are another field that may take advantage of a more constrained subclass relation. In fact, in such a field it is immediate to see the benefits of having a language that ensures that subclasses does not violate the space and time constraints of the superclass.

In the future we plan to extend the present work to cope with multiple inheritance and Java interfaces, too. The first extension is quite straightforward. The case of interfaces is more difficult: an interface is essentially a type specification, though most of the time such a specification is not expressive enough. Consider for example the case of a Java thread, which can be defined using either the `Runnable` interface or the `Thread` class. In both cases the class implementing a thread needs to define a method `run`. So what is the difference between a class implementing `Runnable` or extending `Thread`? The intuition is that in both cases the behavioral type of the class is the same, the difference being just syntactic We plan to define a specification language in order to cope with not-typing properties able to express properties imposed by interfaces. Then we will use it to prove that a class correctly implements an interface.

Acknowledgments

References

[1] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 51–67. Springer-Verlag, 1984.

[2] P. Cousot. Types as abstract interpretations, invited paper. In *POPL '97*, pages 316–331. ACM Press, January 1997.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM Press, January 1977.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*, pages 269–282. ACM Press, January 1979.

[5] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[6] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[7] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.

[8] F. Logozzo. Class-level modular analysis for object oriented languages. In *SAS '03*, volume 2694 of *LNCS*. Springer-Verlag, June 2003.

[9] F. Logozzo. Automatic inference of class invariants. In *VMCAI '04*, volume 2937 of *LNCS*. Springer-Verlag, January 2004.

[10] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.