

# Semantic Class Hierarchies by Abstract Interpretation

Francesco Logozzo  
École Polytechnique  
F-91128 Palaiseau cedex (France)  
[Francesco.Logozzo@polytechnique.fr](mailto:Francesco.Logozzo@polytechnique.fr)

Agostino Cortesi \*  
Università Ca' Foscari di Venezia  
I-30170 Venezia (Italy)  
[cortesi@dsi.unive.it](mailto:cortesi@dsi.unive.it)

## Abstract

A generic semantic-based framework is presented for the definition and manipulation of class hierarchies for object-oriented languages. The framework is based on the notion of observable of a class, i.e. an abstraction of its semantics when focussing on a behavioral property of interest. By exploiting such a notion, we define the semantic subclass relation, capturing the fact that a subclass preserves the behavior of its superclass up to a given (tunable) observed property. We study the relation between syntactic subclass, as present in mainstream object-oriented languages, and the notion of semantic subclass. Such an approach is then extended to class hierarchies, leading to a semantic-based modular treatment of a suite of basic observable-preserving operators on hierarchies. We instantiate the framework by presenting effective algorithms that compute a semantic superclass for two given classes, that extend a hierarchy with a new class, and that merge two hierarchies by preserving semantic subclass relations.

**Keywords:** Abstract Interpretation, Abstract Domains, Inheritance, Static Analysis, Object Oriented Programming

## 1 Introduction

In the object oriented paradigm, a crucial rôle is played by the notion of class hierarchy, as it allow programmers to write code that fulfils valuable quality factors like extensibility and reusability. The underlying idea is to strike commonalities that exist within classes, while accounting for the differences that characterize individual cases [23]. Being a class subclass of another captures the fact that the state and the behavior of the elements of the first one are coherent with the intended meaning

---

\*This work was performed while the author was visiting the École Polytechnique.

of the latter, while disregarding the additional features and functionalities that just characterize the subclass.

The approach of mainstream object oriented languages, like Java and C++, to class hierarchies can be seen as merely syntactic: roughly in such a view hierarchies are collections of classes ordered by the relation “*it has at least the same methods and interface*”. This is why also the main theoretical and practical contributions to hierarchy refactoring issues [29, 30] combine static and dynamic analysis that focus only on syntactic elements. However, as pointed out by [26], this syntactic approach has severe limits, as it leads to troubles when trying to face the issue of extending a given class hierarchy.

In this paper we present a semantic based framework for the definition and manipulation of class hierarchies. It benefits from previous works on abstract interpretation theory [8], that allows to formalize the notion of different levels of property abstraction and of abstract semantics. Our approach is based on the notion of observable of a class, i.e. an abstraction of its semantics when focussing on a behavioral property of interest. The intuition behind is that the semantics of a class can be abstracted by parameterizing it with respect to a given domain of observables, and that a notion of semantic subclass can then be defined in terms of preservation of the observable.

This notion of semantic subclass can be seen as a proper generalization of the concept of class subtyping, having the advantage of being tunable with respect to a given underlying abstract domain and hence of the properties we are interested to capture.

The notion of syntactic subclass, forcing that fields and methods have the same names, is too weak to state something about semantic subclassing, but compatibility results on the syntactic extension on one hand, and suitable renaming functions on the other can be stated that allow us to properly relate the two subclass relations.

The interest of the notion of semantic subclass be-

come even more interesting when facing the problem of manipulating class hierarchies, whose cardinality is often greater than thousands of classes (for instance, NetBeans[25] is made up of 8328 classes). We formalize the notion of semantic ordering of hierarchies: “*when is it the case that a hierarchy is more informative with respect to a give observable?*” As an interesting instantiation of our theoretical framework, we design algorithms to prune hierarchies, to extend a hierarchy with a new class, and to merge two hierarchies by preserving the semantic subclass relation. All these algorithms are effective as soon as the underlying domain of observable is suitable for a static analysis, i.e. its elements are computer representable and it is endowed with a widening operator that guarantees convergence of fixpoint computation.

The structure of the paper is as follows. In Section 2 an example is shown to introduce the main ideas developed in the paper on a suite of simple classes of integers. Some preliminary background on abstract interpretation are presented in Section 3. In Section 4 we introduce the concrete semantics and the notion of class invariant, while Section 5 focusses on semantic and syntactic subclass relations. Results about properties of the ordering among classes are discussed in Section 6, where an algorithm is introduced that computes the semantic subclass relation under suitable hypothesis on the domain of observables. In Section 7 the framework is lifted to hierarchies and in Section 8 some operators on class hierarchies are introduced and proved correct. Finally, Sections 9 and 10 conclude the paper with a discussion on related works and the conclusions, respectively.

## 2 Introductory Example

Let us consider the five classes described in Fig. 1. Intuitively, they encode different sets of integer numbers. So, for instance **Even** is such that the instance variable **x** can only take even values whereas the instance value of **Odd** takes odd values only. In addition, the instance variable of **MultEight** and **MultTwelve** can only take a value that is a multiple of 8 and 12, respectively.

A first question we address is “*Which are the admissible hierarchies between such classes?*”. A hierarchy is admissible when the subclasses preserve a given property of their superclass. So, e.g. when the *parity* of the value taken by the instance field **x** is observed, both the class hierarchies  $\mathcal{H}_1$  and  $\mathcal{H}_2$  in Fig. 2 are admissible. In fact, for  $\mathcal{H}_1$  this is true as the value of **MultEight.x** is always a multiple of 8, in particular it is even. As a consequence, **MultEight** preserves the behavior of **Even** when parity is observed. On the other hand,  $\mathcal{H}_2$  is an admissible class hierarchy too wrt parity. In fact, the

values of **MultTwelve.x** are even numbers, and the values of **MultEight.x** are even numbers too. As a consequence, **MultTwelve** preserves the parity behavior of its superclass **MultEight**. It is worth noting that in such a view, it is also admissible to have **MultEight** subclass of **MultTwelve**. Nevertheless, when we consider a more refined property, for instance the value taken by **x** up to a numerical *congruence*, then  $\mathcal{H}_2$  is no more an admissible hierarchy. In fact, it is immediate to prove that in general a multiple of 12 is not a multiple of 8, so that **MultTwelve** does not preserve the congruence property of its ancestor **MultEight**.

In the example of  $\mathcal{H}_1$  we have that **Even** is a superclass for both **MultEight** and **MultTwelve**. But does it exist a most *precise* superclass? And, if it exists, is it possible to derive it automatically? For instance, in the running example, when the congruences are observed, the class

```
class MultFour {
    int x;
    init() { x = 0 }
    add() { x += 4 }
    sub() { x -= 4 }
}
```

is a superclass for both **MultEight** and **MultTwelve** as the two classes preserve the property that the value of the field **x** is always a multiple of 4. Nevertheless, in general the *most* precise superclass does not exist, as

```
class MultFour_1 {           class MultFour_2 {
    int x;
    init() { x = 0 }
    add() { x += 8 }
    sub() { x -= 4 }
}
class MultFour_2 {
    int x;
    init() { x = 0 }
    add() { x += 4 }
    sub() { x -= 8 }
}
```

are both superclasses of **MultEight** and **MultTwelve**. In fact, there exists infinitely many different *syntactic* classes that have the same *semantic* property (in our example, having a field that is a multiple of 4). At this aim we will introduce the notion of semantic superclass up to an equivalence relation over the *observable* properties of the class.

Such a semantic approach allows also to address the issue of class hierarchies manipulation, by considering for instance the operators for augmenting and merging hierarchies. So for instance, let us consider the class

```
class MultTwenty {
    int x;
    init() { x = 0 }
    add() { x += 20 }
    sub() { x -= 60 }
}.
```

```

class Integer {   class Even {      class Odd {       class MultEight{ class MultTwelve{
int x;           int x;           int x;           int x;           init(){ x = 0 }   init(){ x = 0 }   init(){ x = 1 }   init(){ x = 0 }   init(){ x = 0 }
init(){ x = 0 }   add() { x += 1 }   add() { x += 2 }   add() { x += 2 }   add() { x += 16 }  add() { x += 24 }
add() { x += 1 }   sub() { x -= 1 }   sub() { x -= 2 }   sub() { x -= 2 }   sub() { x -= 8 }   sub() { x -= 12 }
}                   }                   }                   }                   }
}                   }                   }                   }                   }

```

Figure 1: The paper running examples

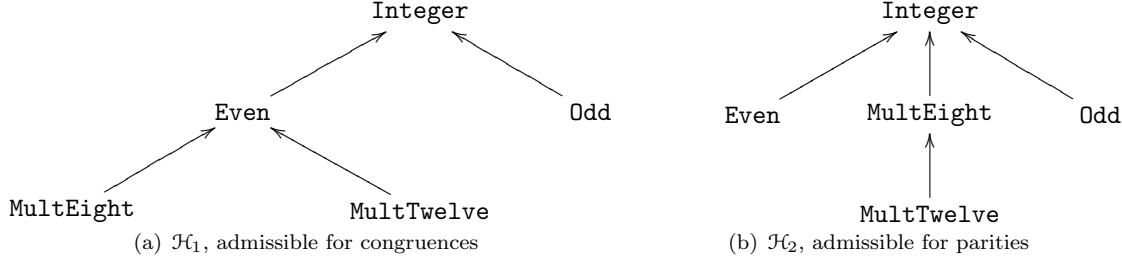


Figure 2: Two Possible Class Hierarchies

When **MultTwenty** is added to the hierarchy  $\mathcal{H}_1$  we obtain the hierarchy  $\mathcal{H}_3$  depicted in Fig. 3(a). It is worth noting that  $\mathcal{H}_3$  is an admissible class hierarchy for congruences. Finally, if we insert **MultFour** into  $\mathcal{H}_3$  we obtain the hierarchy  $\mathcal{H}_4$  in Fig. 3(b), which is admissible for congruences too.

### 3 Preliminaries

Abstract interpretation [10] formalizes the approximation correspondence between the concrete semantics  $\$[\![P]\!]$  of a syntactically correct program  $P$  and an abstract semantics  $\bar{\$}[\![P]\!]$  which is a safe approximation on the concrete semantics. The concrete semantics belongs to a concrete semantic domain  $D$  which is a partially ordered set  $\langle D, \sqsubseteq \rangle$ . An element of  $D$  is a property and the partial order  $\sqsubseteq$  formalizes the loss of information, e.g. the logical implication. The abstract semantics also belongs to a partial order  $\langle \bar{D}, \bar{\sqsubseteq} \rangle$ , which is ordered by the abstract version  $\bar{\sqsubseteq}$  of the concrete approximation order  $\sqsubseteq$ . In this paper we assume that both the concrete and the abstract domain are complete lattices.

The correspondence between the concrete and the abstract semantic domains is given by a pair of functions  $\langle \alpha, \gamma \rangle$  such that  $\alpha \in [\langle D, \sqsubseteq \rangle \rightarrow \langle \bar{D}, \bar{\sqsubseteq} \rangle]$  and  $\gamma \in [\langle \bar{D}, \bar{\sqsubseteq} \rangle \rightarrow \langle D, \sqsubseteq \rangle]$ . The functions  $\alpha$  and  $\gamma$  are called respectively the abstraction and the concretization function. The concretization  $\gamma(\bar{\$}[\![P]\!])$  of the abstract semantics  $\bar{\$}[\![P]\!]$  expresses the abstract information available about program execution in concrete terms. It should be a sound approximation of the concrete semantics, i.e.  $\$[\![P]\!] \sqsubseteq \gamma(\bar{\$}[\![P]\!])$ .

If any element  $d$  of the concrete domain  $D$  has a best approximation (i.e.  $\bar{\sqsubseteq}$ -most precise) in the abstract domain  $\bar{D}$  given by  $\alpha(d)$ , then the pair  $\langle \alpha, \gamma \rangle$  is called a Galois connection [13]. This is written as

$$\langle D, \sqsubseteq \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \bar{D}, \bar{\sqsubseteq} \rangle.$$

By definition, this means that

$$\forall d \in D. \forall \bar{d} \in \bar{D}. \alpha(d) \bar{\sqsubseteq} \bar{d} \iff d \sqsubseteq \gamma(\bar{d}).$$

Galois connections enjoy several properties [12]. First, they can be composed, so that the composition of two Galois connections is still a Galois connection: if

$$\langle D, \sqsubseteq \rangle \xrightleftharpoons[\alpha_1]{\gamma_1} \langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle \text{ and } \langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle \xrightleftharpoons[\alpha_2]{\gamma_2} \langle \bar{D}, \bar{\sqsubseteq} \rangle$$

then

$$\langle D, \sqsubseteq \rangle \xrightleftharpoons[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} \langle \bar{D}, \bar{\sqsubseteq} \rangle.$$

Second, one function in a Galois connection uniquely determines the other:

$$\begin{aligned} \alpha(d) &= \cap \{ \bar{d} \mid d \sqsubseteq \gamma(\bar{d}) \} \\ \gamma(\bar{d}) &= \cup \{ d \mid \alpha(d) \bar{\sqsubseteq} \bar{d} \} \end{aligned}$$

so that in the following we often omit one of the twos.

A well-known result in abstract interpretation theory [11] is that if the concrete domain  $\langle D, \sqsubseteq \rangle$  is a complete lattice, then the set of all its abstractions, i.e.

$$\mathcal{A}(D) = \{ \langle \bar{D}, \bar{\sqsubseteq} \rangle \mid \exists \langle \alpha, \gamma \rangle. \langle D, \sqsubseteq \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \bar{D}, \bar{\sqsubseteq} \rangle \}$$

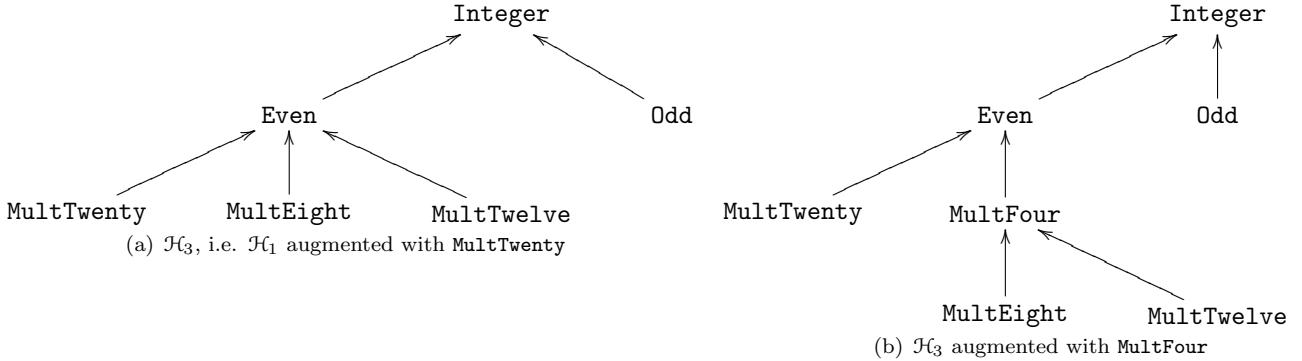


Figure 3: Insertions to the hierarchy  $\mathcal{H}_1$

is a complete lattice w.r.t. the order relation  $\leq$  that intuitively says that a domain is more precise than another. Formally:

$$\langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle \leq \langle \bar{D}_2, \bar{\sqsubseteq}_2 \rangle \iff \exists \langle \alpha, \gamma \rangle. \langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \bar{D}_2, \bar{\sqsubseteq}_2 \rangle.$$

The least element of  $\mathcal{A}(D)$ , i.e. the most precise domain, is the concrete one  $\langle D, \sqsubseteq \rangle$  and the largest, i.e. the least precise domain, is  $\langle \{\top\}, \text{id} \rangle$ , where  $\top$  is the largest element of  $\langle D, \sqsubseteq \rangle$  and  $\text{id}$  is the identity relation. Given two domains  $\langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle, \langle \bar{D}_2, \bar{\sqsubseteq}_2 \rangle \in \mathcal{A}(D)$ , the *reduced product* is the greatest domain more precise than both, hence it is  $\langle \bar{D}_1, \bar{\sqsubseteq}_1 \rangle \wedge \langle \bar{D}_2, \bar{\sqsubseteq}_2 \rangle$ .

Given a concrete domain  $\langle D, \sqsubseteq \rangle$ , an abstract domain  $\langle \bar{D}, \bar{\sqsubseteq} \rangle$ , a concrete semantic function  $s[\mathbb{P}] \in [D \rightarrow D]$  and a Galois connection  $\langle \alpha, \gamma \rangle$  the *best abstract function*  $\bar{s}[\mathbb{P}] \in [\bar{D} \rightarrow \bar{D}]$  is  $\bar{s}[\mathbb{P}] = \alpha \circ s[\mathbb{P}] \circ \gamma$ .

## 4 Class Semantics and Analysis

The concrete semantics describes the properties of the execution of programs. The goal of a static analysis is to provide an effective computable approximation of the concrete semantics [10]. Therefore, the first step for the design of a static analysis is the definition of the concrete semantics.

### 4.1 Syntax

A class is a template for objects. It is provided by the programmer that specifies the fields, the methods and the class constructor. Then it can be abstractly modelled by a triplet, as in the following (simplified) definition:

**Definition 1 (Classes)** A class  $C$  is a triplet  $\langle F, \text{init}, M \rangle$  where  $F$  is a set of distinct variables,  $\text{init}$  is the class constructor and  $M$  is a set of function definitions. The set of all the classes is denoted by  $\mathcal{C}$ .

It is worth noting that for the sake of generality in the definition above we do not ask to have typed fields or methods. Moreover we assume that all the class fields are protected. This is just to simplify the exposition, and it does not cause any loss of generality: in fact any external access to a field  $f$  can be simulated by a couple of methods `set_f`/ `get_f`. Furthermore we assume to have a single class constructor. The generalization to an arbitrary number of constructors is straightforward.

The interface of a class is the set of names exposed by it:

**Definition 2 (Class Interface)** Given a class  $C = \langle F, \text{init}, M \rangle$ , let  $M_{\text{names}}$  be the names of the  $C$ 's methods. Then the interface of  $C$  is  $\iota(C) = F \cup \{\text{init}\} \cup M_{\text{names}}$ .

### 4.2 Concrete Semantics

Given a class  $C = \langle F, \text{init}, M \rangle$ , every instance of  $C$  has an internal state  $\sigma \in \Sigma$  that is a function from fields to values, i.e.  $\Sigma = [F \rightarrow D_{\text{val}}]$ .  $D_{\text{val}}$  is the semantic domain of values.

When a class is instantiated, for example by means of the `new` construct, the class constructor is called to set up the new object internal state. This can be modelled by a semantic function  $i[\text{init}] \in [D_{\text{in}} \rightarrow \wp(\Sigma)]$ , where  $D_{\text{in}}$  is the semantic domain for the constructor input values (if any). We consider sets in order to model non-determinism, e.g. user input or random choices.

The semantics of a method  $m$  is a function  $m[\mathbb{m}] \in [D_{\text{in}} \times \Sigma \rightarrow \wp(D_{\text{out}} \times \Sigma)]$ . Indeed a method is called with two parameters: the method actual parameters and the internal state of the object it belongs to. The output of a method is a set of pairs  $\langle$  return value (if any), new object state  $\rangle$ .

The most precise property of a class  $C$  is the set of states reached by any execution of every instance of  $C$  in any possible context. Stated otherwise, the most precise class property are the states reached in any possible computation of one of its instances.

Such a property can be expressed as a least fixpoint on the complete boolean lattice  $\langle \mathcal{P}(\Sigma), \subseteq \rangle$  as follows. The set of the initial states, i.e. the states reached after any invocation of the  $C$  constructor, is:

$$S_0 = \{\sigma \mid \exists v \in D_{in}. \sigma \in \mathbb{I}[\text{init}](v)\}.$$

The states reached after the invocation of a method  $m$  are given by the method collecting *forward* semantics  $M^>[m] \in [\wp(\Sigma) \rightarrow \wp(\Sigma)]$ :

$$\begin{aligned} M^>[m](S) = \{\sigma' \in \Sigma \mid &\exists \sigma \in S. \exists v \in D_{in}. \\ &\exists v' \in D_{out}. \langle v', \sigma' \rangle \in m[m](v, \sigma)\}. \end{aligned}$$

The class reachable states are a solution of the following system of recursive equation:

$$\begin{aligned} S &= S_0 \cup \bigcup_{m \in M} S_m \\ S_m &= M^>[m](S) \quad m \in M. \end{aligned} \tag{1}$$

The former equation characterizes, according to the intuition, the set of states that are reachable before and after the execution of any method in any instance of the class<sup>1</sup>. In particular, the least solution of (1) w.r.t. the set inclusion is a tuple  $\langle S, S_0, \{m : S_m\} \rangle$  such that  $S$  is a class invariant [22], and for each method  $m$ ,  $S_m$  is the strongest postcondition of the method. The methods preconditions can be obtained going backward from the postconditions, i.e. given a method  $m$  and its postcondition we consider the set of states from which it is possible to reach a state in  $S_m$  by an invocation to  $m$ . Formally, the collecting *backward* method semantics  $M^<[m] \in [\wp(\Sigma) \rightarrow \wp(\Sigma)]$  is defined as

$$\begin{aligned} M^<[m](S) = \{\sigma \in \Sigma \mid &\exists \sigma' \in S. \exists v \in D_{in}. \\ &\exists v' \in D_{out}. \langle v', \sigma' \rangle \in m[m](v, \sigma)\}. \end{aligned}$$

and the methods preconditions are  $B_m = M^<[m](S_m)$ . Therefore the concrete class semantics, i.e. the most precise property of a class, is the triplet

$$C[C] = \langle S, S_0, \{m : B_m \rightarrow S_m\} \rangle. \tag{2}$$

The use of the concrete semantics  $C[C]$  for the definition of the observables of a class has two drawbacks. First, in general the computation of the least fixpoint of (1) may be unfeasible and the sets  $S$  and  $S_m$  and  $B_m$  may not be computer-representable. Therefore, such an approach it is not suitable for an effective definition of semantic subclassing. Second, it is too precise, so that it may differentiate between classes that do not need to be

<sup>1</sup>In general, the body of a method  $m_i$  may invoke a method  $m_j$  that belongs to the same class. However, as such a call is somehow private to the class at this point the class invariant is not required to hold [23].

distinguished. For example, let us consider two classes `StackWithList` and `StackWithArray` which implement a stack by using respectively a linked list and a resizable array. Both of them have a method `push` to push an element on the top stack and one method `pop` to return and remove the top element. If they are observed using the concrete semantics, then the two classes are unrelated as the internal representation of the stack is different. On the other hand, when the behavior w.r.t. to the invocation of methods is observed, they act in the same way, e.g. no difference can be made between the values returned by the implementation of `pop` made by `StackWithList` and `StackWithArray`: both of them return the value on the top of the stack.

In order to overcome those drawbacks we consider abstract domains which encode the relevant properties and abstract semantics that are feasible, namely which are sound, but not necessarily complete, abstractions of the concrete semantics.

### 4.3 Domain of Observables

An observable of a class  $C$  is an approximation of its semantics that captures some aspects of interest of the behavior of  $C$ . In this section we show how to build a domain of observables starting from an abstraction of sets of object states.

Let us consider an abstract domain  $\langle P, \sqsubseteq \rangle$  which approximates sets of object states, i.e. such that

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \xrightleftharpoons[\alpha]{\gamma} \langle P, \sqsubseteq, \perp, \top, \sqcup, \sqcap \rangle. \tag{3}$$

The intuition behind such a domain is that it captures the properties we are interested in, while abstracting away the others we do not care. For instance, if we are interested in the linear relations between the values of the fields of  $C$  instances we instantiate  $P$  with the Polyhedra abstract domain [15]. On the other hand if the interest is on object aliasing then we are likely to chose for  $P$  an abstract domain that captures shapes, e.g. the one presented in [27].

Once fixed  $\langle P, \sqsubseteq \rangle$ , the abstract domain  $\langle O[P], \sqsubseteq_o^{[P]} \rangle$  of the observables of a class is built on the top of it. The elements of the abstract domain belong to the set:

$$O[P] = \{(\bar{S}, \bar{S}_0, \{m : (\bar{V}_m, \bar{B}_m) \rightarrow \bar{S}_m\}) \mid \bar{S}, \bar{S}_0, \bar{V}_m, \bar{B}_m, \bar{S}_m \in P\}.$$

Intuitively, an element of  $O[P]$  consists in an approximation of the class invariant, the constructor postcondition, and for each method an approximation of its precondition and postcondition. At its turn, a method precondition is made up of two parts, one for the method input values and the other for the object internal state.

When clear from the context, we write  $\langle O, \sqsubseteq_o \rangle$  instead of  $\langle O[P], \sqsubseteq_o^{[P]} \rangle$ .

We tacitly assume that if a method  $\mathbf{o}$  is not defined in a class, then its precondition and postconditions are respectively  $\top$  and  $\perp$ .

The order  $\sqsubseteq_o$  on  $O$  is defined pointwise. Let  $\mathbf{o}_1 = \langle \bar{I}, \bar{I}_0, \{\mathbf{m}_i : \langle \bar{U}_i, \bar{R}_i \rangle \rightarrow \bar{I}_i\} \rangle$  and  $\mathbf{o}_2 = \langle \bar{J}, \bar{J}_0, \{\mathbf{m}_j : \langle \bar{W}_j, \bar{Q}_j \rangle \rightarrow \bar{J}_j\} \rangle$  be two elements<sup>2</sup> of  $O$ . Then the order  $\sqsubseteq_o$  is defined as:

$$\mathbf{o}_1 \sqsubseteq_o \mathbf{o}_2 \iff \bar{I} \sqsubseteq \bar{J} \wedge \bar{I}_0 \sqsubseteq \bar{J}_0 \wedge \\ (\forall \mathbf{m}_i. \bar{W}_i \sqsubseteq \bar{U}_i \wedge \bar{R}_i \sqsubseteq \bar{Q}_i \wedge \bar{I}_i \sqsubseteq \bar{J}_i).$$

Roughly speaking, if  $\mathbf{o}_1$  and  $\mathbf{o}_2$  are the observables of two classes A and B then the order  $\sqsubseteq_o$  ensures that A preserves the class invariant of B and that the methods of A are a “safe” replacement of those with the same name in B. Intuitively, the precondition condition generalizes the observations, made in the context of type theory, of [4]. It states two things. First, if the context satisfies  $\bar{W}_i$  then it satisfies the inherited method precondition  $\bar{U}_i$  too (i.e.  $\bar{W}_i \sqsubseteq \bar{U}_i$ ). Thus the inherited method can be used in any context where its ancestor can. Second, the state of  $\mathbf{o}_1$  before the invocation of a method must be *compatible* with that of  $\mathbf{o}_2$  (i.e.  $\bar{R}_i \sqsubseteq \bar{Q}_i$ ). Finally, the postcondition of the inherited method may be stronger than that of the ancestor (i.e.  $\bar{I}_i \sqsubseteq \bar{J}_i$ ).

Having defined  $\sqsubseteq_o$ , it is routine to check that, as  $P$  is a complete lattice,  $\perp_o = \langle \perp, \perp, \{\mathbf{m}_i : \langle \top, \perp \rangle \rightarrow \perp\} \rangle$  is the smallest element of  $O$  and  $\top_o = \langle \top, \top, \{\mathbf{m}_i : \langle \perp, \top \rangle \rightarrow \top\} \rangle$  is the largest one.

The join and the meet operations on observables are defined point-wise. Let  $\mathbf{o}_1 = \langle \bar{I}, \bar{I}_0, \{\mathbf{m}_i : \langle \bar{U}_i, \bar{R}_i \rangle \rightarrow \bar{I}_i\} \rangle$  and  $\mathbf{o}_2 = \langle \bar{J}, \bar{J}_0, \{\mathbf{m}_j : \langle \bar{W}_j, \bar{Q}_j \rangle \rightarrow \bar{J}_j\} \rangle$  be two elements of  $O$ . Then

$$\begin{aligned} \mathbf{o}_1 \sqcup_o \mathbf{o}_2 &= \langle \bar{I} \sqcup \bar{J}, \bar{I}_0 \sqcup \bar{J}_0, \{\mathbf{m}_i : \langle \bar{U}_i \sqcup \bar{W}_i, \bar{Q}_i \sqcup \bar{R}_i \rangle \rightarrow \bar{I}_i \sqcup \bar{J}_i \} \rangle \\ \mathbf{o}_1 \sqcap_o \mathbf{o}_2 &= \langle \bar{I} \sqcap \bar{J}, \bar{I}_0 \sqcap \bar{J}_0, \{\mathbf{m}_i : \langle \bar{U}_i \sqcap \bar{W}_i, \bar{Q}_i \sqcap \bar{P}_i \rangle \rightarrow \bar{I}_i \sqcap \bar{J}_i \} \rangle. \end{aligned}$$

Moreover, let us suppose that the order relation  $\sqsubseteq$  is decidable. For instance, this is the case of an abstract domain used for an effective static analysis. As  $\sqsubseteq_o$  is defined in terms of  $\sqsubseteq$  and the universal quantification ranges on a finite number of methods then  $\sqsubseteq_o$  is decidable too. To sum up the results of this section we have the

**Theorem 1** *Let  $\langle P, \sqsubseteq, \perp, \top, \sqcup, \sqcap \rangle$  be a complete lattice. Then  $\langle O, \sqsubseteq_o, \perp_o, \top_o, \sqcup_o, \sqcap_o \rangle$  is a complete lattice. Moreover, if  $\sqsubseteq$  is decidable then  $\sqsubseteq_o$  is decidable too.*

Galois connections can be lifted to the domain of observables:

<sup>2</sup>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  $\mathbf{m}_i$  of  $\mathbf{o}_1$  and  $\mathbf{o}_2$ .

**Lemma 1** *Let  $\langle P, \sqsubseteq \rangle$  and  $\langle P', \sqsubseteq' \rangle$  be two domains in  $\mathcal{A}(\mathcal{P}(\Sigma))$  such that  $\langle P, \sqsubseteq \rangle \leq \langle P', \sqsubseteq' \rangle$  with the Galois connection  $\langle \alpha, \gamma \rangle$ . Then,*

$$\langle O[P], \sqsubseteq_o^{[P]} \rangle \xrightleftharpoons[\alpha_o]{\gamma_o} \langle O[P'], \sqsubseteq_o^{[P']} \rangle$$

where  $\alpha_o$  and  $\gamma_o$  are

$$\begin{aligned} \alpha_o(\langle \bar{S}, \bar{S}_0, \{\mathbf{m} : \langle \bar{V}_m, \bar{B}_m \rangle \rightarrow \bar{S}_m \} \rangle) &= \langle \alpha(\bar{S}), \alpha(\bar{S}_0), \\ &\quad \{\mathbf{m} : \langle \alpha(\bar{V}_m), \alpha(\bar{B}_m) \rangle \rightarrow \alpha(\bar{S}_m) \} \rangle \\ \gamma_o(\langle \bar{S}', \bar{S}'_0, \{\mathbf{m} : \langle \bar{V}'_m, \bar{B}'_m \rangle \rightarrow \bar{S}'_m \} \rangle) &= \langle \gamma(\bar{S}), \gamma(\bar{S}'_0), \\ &\quad \{\mathbf{m} : \langle \gamma(\bar{V}'_m), \gamma(\bar{B}'_m) \rangle \rightarrow \gamma(\bar{S}'_m) \} \rangle. \end{aligned}$$

The lemma above and the fact that  $\mathcal{A}(\mathcal{P}(\Sigma))$  is a complete lattice, imply that:

**Lemma 2 (Most Concrete Domain of Observables)**  *$O[\mathcal{P}(\Sigma)]$  is the most precise domain of observables.*

#### 4.4 Abstract Semantics

Once the abstract domain is settled, an abstraction of  $\mathbb{C}[\mathbb{C}]$  is obtained by considering the abstract counterpart of (1). As a first step we need to consider the abstract counterparts for the initial states, the forward and the backward collecting semantics. For the sake of simplicity, we consider the *best* abstract counterparts of such concrete semantic functions. Nevertheless, in general any upper approximation would be a sound approximation.

The best approximation for the class initial states is the image of  $S_0$  through  $\alpha$ , i.e.  $\alpha(S_0) = \bar{S}_0$ .

The best approximation in  $P$  of the forward collecting method semantics of a method  $\mathbf{m}$  of  $\mathbb{C}$  is  $\bar{M}^>[\mathbf{m}] \in [P \rightarrow P]$  defined as

$$\bar{M}^>[\mathbf{m}](\bar{S}) = \alpha \circ M^>[\mathbf{m}] \circ \gamma(\bar{S}).$$

The abstract counterpart of (1) is the following equation system:

$$\begin{aligned} \bar{S} &= \bar{S}_0 \sqcup \bigsqcup_{\mathbf{m} \in \mathbb{M}} \bar{S}_{\mathbf{m}} \\ \bar{S}_{\mathbf{m}} &= \bar{M}^>[\mathbf{m}](\bar{S}) \quad \mathbf{m} \in \mathbb{M}. \end{aligned} \tag{4}$$

It is immediate to see that the above equations are monotonic and, because of the Tarski fixpoint theorem, (4) has a least solution  $\langle \bar{S}, \bar{S}_0, \{\mathbf{m} : \bar{S}_{\mathbf{m}}\} \rangle$ . Similarly to the concrete case, the abstract preconditions can be obtained by considering the best approximation of the backward collecting method semantics  $\bar{M}^<[\mathbf{m}] \in [P \rightarrow P]$  defined as

$$\bar{M}^<[\mathbf{m}](\bar{S}) = \alpha \circ M^<[\mathbf{m}] \circ \gamma(\bar{S}). \tag{5}$$

The method abstract preconditions are obtained by projecting  $\bar{M}^<[\mathbf{m}](\bar{S}_{\mathbf{m}})$  respectively on the method input

values and the instance fields. Therefore the method abstract preconditions are

$$\begin{aligned}\bar{V}_m &= \pi_{in}(\bar{M}^{\leq}[\![m]\!](\bar{S}_m)) \\ \bar{B}_m &= \pi_F(\bar{M}^{\leq}[\![m]\!](\bar{S}_m)).\end{aligned}$$

To sum up, the triplet

$$\bar{C}[\![C]\!] = \langle \bar{S}, \bar{S}_0, \{m : \langle \bar{V}_m, \bar{B}_m \rangle \rightarrow \bar{S}_m\} \rangle \quad (6)$$

belongs to the domain of observables, and it is a sound approximations of the semantics of  $C$ , w.r.t the properties encoded by the abstract domain  $\langle P, \sqsubseteq \rangle$ . Formally:

**Theorem 2 (Soundness)** *Let  $C$  be a class and let  $C[\![C]\!]$  be the most precise property of  $C$  as in (2). Furthermore let  $\langle P, \sqsubseteq \rangle$  be an abstract domain which satisfies (3), and let  $\bar{C}[\![C]\!]$  be as in (6). Then*

$$\alpha_o(C[\![C]\!]) \sqsubseteq_o \bar{C}[\![C]\!].$$

Roughly speaking, the theorem above means that when a given property of a class is observed, then  $\bar{C}[\![C]\!]$  is a sound approximation of the behavior of  $C$ . Therefore, it is possible to define the observable of a class as its abstract semantics:

**Definition 3 (Observable of a Class)** *The observable of a class  $C$  w.r.t. the property encoded by  $\langle P, \sqsubseteq \rangle$  is*

$$\bar{C}[\![C]\!] = \langle \bar{S}, \bar{S}_0, \{m : \langle \bar{V}_m, \bar{B}_m \rangle \rightarrow \bar{S}_m\} \rangle.$$

**Example 1** *Let us instantiate  $\langle P, \sqsubseteq \rangle$  with  $Con$ , the Granger's abstract domain of equalities of linear congruences [19]. The elements of such a domains are in the form  $x \equiv a(b)$ , where  $x$  is a program variable and  $a$  and  $b$  are integers. The meaning is expressed by a  $\gamma_c \in [Con \rightarrow \mathcal{P}(\Sigma)]$  defined as*

$$\gamma_c(x \equiv a(b)) = \{\sigma \in \Sigma \mid \exists k \in \mathbb{N}. \sigma(x) = a + k \cdot b\}.$$

*Let us consider the classes  $Even$  and  $MultEight$  in Fig. 2 and let  $e$  be the property  $x \equiv 0(2)$ ,  $d$  the property  $x \equiv 1(2)$  and  $u$  be the property  $x \equiv 0(8)$ . Then the observables of  $Even$  and  $MultEight$  w.r.t.  $Con$  are*

$$\begin{aligned}\bar{C}[\![Even]\!] &= \langle e, e, \{\text{add} : \langle \perp, e \rangle \rightarrow e, \text{sub} : \langle \perp, e \rangle \rightarrow e\} \rangle \\ \bar{C}[\![0dd]\!] &= \langle d, d, \{\text{add} : \langle \perp, d \rangle \rightarrow d, \text{sub} : \langle \perp, d \rangle \rightarrow d\} \rangle \\ \bar{C}[\![MultEight]\!] &= \langle u, u, \{\text{add} : \langle \perp, u \rangle \rightarrow u, \text{sub} : \langle \perp, u \rangle \rightarrow u\} \rangle\end{aligned}$$

*It is worth noting that as  $\text{add}$  and  $\text{sub}$  do not have an input parameter, the corresponding precondition for the input values is  $\perp$ .*

## 5 Subclassing

We consider two facets of subclassing: syntax and semantics. Roughly, given two classes  $A$  and  $B$ ,  $A$  is a *syntactic* subclass of  $B$ ,  $A \blacktriangleleft B$ , if all the names defined in  $B$  are defined in  $A$  too. On the other hand,  $A$  is a *semantic* subclass of  $B$ ,  $A \triangleleft B$  if  $A$  preserves the observable of  $B$ .

### 5.1 Syntactic Subclassing

The intuition behind the syntactic subclassing relation is inspired by the Smalltalk [17] understanding of inheritance: a subclass may answer to all the messages sent to its superclass. Stated otherwise, the syntactic subclassing relation is defined in terms of inclusion of class interfaces:

**Definition 4 (Syntactic Subclassing)** *Let  $A$  and  $B$  be two classes. Then the syntactic subclass relation is defined as:*

$$A \blacktriangleleft B \iff \iota(A) \supseteq \iota(B).$$

It is worth noting that as  $\iota(\cdot)$  does not distinguish between names of fields and methods, a class  $A = \langle \emptyset, \text{init}, f = \lambda x. x + 1 \rangle$  is a syntactic subclass of  $B = \langle f, \text{init}, \emptyset \rangle$ , even if in the first case  $f$  is a name of a method and in the second it is the name of a fields. This is meaningful in the general, untyped, context we put ourselves.

**Example 2** *In mainstream object oriented languages the subclassing mechanism is provided through class extension. For example, in Java a subclass of a base class  $B$  is created using the syntactic construct “ $A$  extends  $B$  { extension }”, where  $A$  is the name of the subclass and  $\text{extension}$  are the field and the methods added and/or redefined by the subclass. As a consequence, if type declarations are considered part of the fields and method names, then in Java  $A \blacktriangleleft B$  always holds.*

### 5.2 Semantic Subclassing

The semantic subclassing relation formalizes the intuition that up-to a given property a class  $A$  behaves like  $B$ . For example, if the property of interest is the type of the class, then  $A$  is a semantic subclass of  $B$  if its type is a subtype of  $B$ . In our framework, the semantic subclassing can be defined in terms of preservation of observables. In fact, as  $\sqsubseteq_o$  is the abstract counterpart of the logical implication then  $\bar{C}[\![A]\!] \sqsubseteq_o \bar{C}[\![B]\!]$  means that  $A$  preserves the semantics of  $B$ , when a given property of interest is observed. Therefore we can define

**Definition 5 (Semantic Subclassing)** *Let  $\langle O, \sqsubseteq_o \rangle$  be an abstract domain of observables and let  $A$  and  $B$*

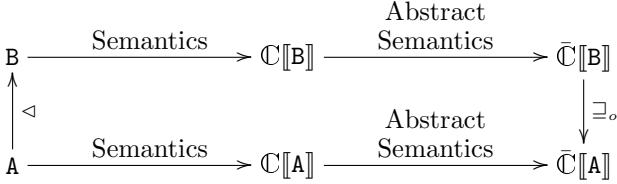


Figure 4: A Visualization of the Semantic Subclassing Relation

be two classes. Then the semantic subclassing relation is defined as:

$$A \triangleleft_O B \iff \bar{C}[[A]] \sqsubseteq_o \bar{C}[[B]].$$

**Example 3** Let us consider the classes Even, Odd and MultEight and their respective observables as in Ex. 1. Then, as  $u \sqsubseteq e$  holds, we have that MultEight  $\triangleleft$  Even. On the other hand, we have that neither  $e \sqsubseteq d$  nor  $d \sqsubseteq e$ . As a consequence, Even  $\not\triangleleft$  Odd and Odd  $\not\triangleleft$  Even.

Observe that when  $\langle O, \sqsubseteq_o \rangle$  is instantiated with the types abstract domain [9] then the relation defined above coincides with the traditional subtyping-based definition of subclassing [3].

The relation between classes, concrete semantics and observables can be visualized by the diagram in Fig. 4. The diagram essentially shows how the concept of semantic 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  $O$ . This is in accord with the mundane understanding of inheritance which states that a subclass is a specialization of the ancestor [23].

Next lemma states a kind of monotony of  $\triangleleft$  w.r.t. the observed properties, in that a less precise abstract domains cannot improve the grain of the relation:

**Lemma 3** Let A and B be two classes,  $\langle P, \sqsubseteq \rangle$  and  $\langle P', \sqsubseteq' \rangle$  be two abstract domains in  $\mathcal{A}(\mathcal{P}(\Sigma))$  such that  $\langle P, \sqsubseteq \rangle \leq \langle P', \sqsubseteq' \rangle$ . Then, if  $A \triangleleft_{O[P]} B$  then  $A \triangleleft_{O[P']} B$ .

A first consequence of Lemma 3 is that the more precise the domain of observables the more precise the induced subclass relation. Roughly, if we “observe” a more precise property about the class semantics then we are able to better “distinguish” between the different classes:

**Corollary 1** Let  $\langle O, \sqsubseteq_o \rangle$  and  $\langle O', \sqsubseteq'_o \rangle$  be two domains of observables such that  $\langle O, \sqsubseteq_o \rangle \leq \langle O', \sqsubseteq'_o \rangle$ . Then  $\triangleleft_O \subseteq \triangleleft_{O'}$ .

**Example 4** Let us consider hierarchies  $\mathcal{H}_1$  and  $\mathcal{H}_2$  depicted in Fig. 2. As the domain of congruences is (strictly) more precise than the domain of parities,  $\mathcal{H}_1$  is also admissible for parities, by Cor. 1. Observe that in general the converse is not true: for instance  $\mathcal{H}_2$  is not admissible for congruences.

In particular, when consider the reduced product of domains it follows that

**Corollary 2** Let A and B two classes, and let  $\langle O, \sqsubseteq_o \rangle$  and  $\langle O', \sqsubseteq'_o \rangle$  be two domains of observables. If  $A \triangleleft_{O \wedge O'} B$  then  $\forall \langle O'', \sqsubseteq''_o \rangle \leq (\langle O, \sqsubseteq_o \rangle \wedge \langle O', \sqsubseteq'_o \rangle). A \triangleleft_{O''} B$ .

The semantic subclass relation induced by the most abstract domain is the trivial one, in which all the classes are in relation with all the others. Differently stated, the most abstract domain does not capture any property, so that the observables of all the classes are smashed to  $\top$ , i.e. the “I do not know”.

**Corollary 3** Let  $P_\top$  be the largest element of  $\mathcal{A}(\mathcal{P}(\Sigma))$ . Then  $\triangleleft_{O[P_\top]} = \{\langle A, B \rangle \mid A, B \in \mathcal{C}\}$ .

When considering the *identity* Galois connection  $\langle \lambda x. x, \lambda x. x \rangle$  Def. 5 boils down to the observation of the concrete semantics, so that by Lemma 3,  $\triangleleft_{O[\mathcal{P}(\Sigma)]}$  is the most precise semantic subclassing relation.

Given two classes A and B such that  $A \triangleleft_O B$ , one may wonder whether it exists a domain of observables  $O'$  such that  $A \triangleleft_{O'} B$ . The answer is “yes”, as shown by the next lemma.

**Lemma 4 (Existence of  $\triangleleft$ )** Let A and B be two classes. Then it exists a domain  $\langle P, \sqsubseteq \rangle$  such that  $A \triangleleft_{O[P]} B$ .

*Proof.* Let  $P_\top = \langle \{\top\}, \text{id} \rangle$  be the largest element of  $\mathcal{A}(\mathcal{P}(\Sigma))$ . Then the best abstract semantics maps all the classes to  $\top$ :  $\forall C \in \mathcal{C}. \bar{C}[[C]] = \top$ . As a consequence,  $\bar{C}[[A]] = \bar{C}[[B]] = \top$  so that  $\bar{C}[[A]] \triangleleft_{O[P_\top]} \bar{C}[[B]]$ . *q.e.d.*

Then, one my wonder if given two classes, there exists a *least* domain of observables such that the two are in the semantic subclass relation. It turns out that the answer to this question is “no”, as shown by the following example:

**Example 5** Let us consider two classes A and B that are equal except for a method m defined, in the case of A as

```
A.m() {
    x = 1; y = 2;
    if (x > 0) && (y % 2 == 0) {
        x = 1; y = 4;
    } else {
        x = 1; y = 8;
    }
}.
```

and in the case of B as

```
B.m() {
  x = 1; y = 2;
  if (x > 0) && (y % 2 == 0) {
    x = 1; y = 2;
  } else {
    x = 3; y = 10;
  }
}
```

When considering the domain of intervals [10] as observables, we infer that  $A \triangleleft_{\text{Intervals}} B$  as

$$([1, 1], [4, 10]) \sqsubseteq ([1, 3], [2, 10])$$

and when considering the domain of parities as observables, we infer that  $A \triangleleft_{\text{Parities}} B$  as

$$(\text{odd}, \text{even}) \sqsubseteq (\text{odd}, \text{even}).$$

In fact, in both cases the abstract domain is not precise enough to capture the branch chosen by the conditional statement. Nevertheless, when considering the reduced product Intervals  $\wedge$  Parities we have that  $A \not\triangleleft_{\text{Intervals} \wedge \text{Parities}} B$  as

$$(([1, 1], \text{odd}), ([4, 4], \text{even})) \not\sqsubseteq(([1, 1], \text{odd}), ([2, 2], \text{even})).$$

As a consequence, if there exists a least domain O such that  $A \triangleleft_O B$ , then O should be strictly smaller than both Intervals and Parities as the two domains are not comparable. Then, O must be smaller or equal to the reduced product of the two domains. We have just shown that it cannot be equal. By Cor. 2 it follows that it cannot be smaller, too.

### 5.3 Relation between $\blacktriangleleft$ and $\triangleleft$

We have introduced two distinguished notions of subclassing, a semantic and a syntactic one. But which is the relation between those two? Does semantic subclassing imply the syntactic one or *vice versa*? Under which conditions? The goal of this section is to study such a relation.

Let us consider two classes A and B such that  $A \blacktriangleleft B$ . By definition this means that all the names (fields or methods) defined in B are defined in A too. However, such a condition is too weak to state something about the semantics of A w.r.t. that of B:

**Example 6** Let us consider the two classes:

$$\begin{aligned} A &= \langle \{x\}, \lambda(). x := 1, \{m = \lambda(). x := 1/x, n = \lambda(). x := 0\} \rangle, \\ B &= \langle \{x\}, \lambda(). x := 0, \{m = \lambda(). x++; x := 1/x\} \rangle. \end{aligned}$$

As the name of methods of B are included in that of A,  $A \blacktriangleleft B$ . Nevertheless, an instance of B will never

perform a division by zero whereas one of A can. As a consequence the behavior of the two classes may be different, so that for all the non-trivial properties,  $A \not\triangleleft B$ .

The last example shows that in general  $A \blacktriangleleft B \not\Rightarrow A \triangleleft B$ . By Lemma 4 there exists a domain of observables O such that  $A \triangleleft_O B$ . Nevertheless, in most cases (e.g. Ex. 6) such a domain is the most abstract one, so that by Cor. 3  $\triangleleft$  is a non-interesting relation. Therefore, in order to obtain more interesting subclass relations, we have to consider some hypotheses on the abstract semantics of the methods of the class. As a matter of fact we have the theorem below which essentially states that if the constructor of a class A is compatible with that of B and if the methods of A does not violate the class invariant of B then A is a semantic subclass of B.

**Theorem 3** Let  $A = \langle F_A, \text{init}_A, M_A \rangle$  and  $B = \langle F_B, \text{init}_B, M_B \rangle$  be two classes such that  $A \blacktriangleleft B$ . Furthermore, let  $\langle P, \sqsubseteq \rangle \in \mathcal{A}(\mathcal{P}(\Sigma))$ . If  $\forall \bar{S} \in P$ .

- $I_B$  is a class invariant for B
- $\bar{M}^>[\text{init}_A] \sqsubseteq \bar{M}^>[\text{init}_B]$
- $\forall m \in M_A \cap M_B. \bar{M}^>[m](\bar{S}) \sqsubseteq I_B$
- $\forall m \in M_A. m \notin M_B \implies \bar{M}^>[m](\bar{S}) \sqsubseteq I_B$

then  $A \triangleleft_{O[P]} B$ .

*Proof.(Sketch)* If  $I_B$  is class invariant, then it is an upper approximation of the least solution of (4). By theorem hypotheses and Tarski's fixpoint theorem, it turns out that the class invariant for A,  $I_A$  is such that  $I_A \sqsubseteq I_B$ , so that A preserves the B invariant. By (4) and (5),  $\bar{C}[A] \sqsubseteq_o \bar{C}[B]$ , which concludes the proof. *q.e.d.*

On the other hand, semantics subclassing almost implies syntactic subclassing:

**Theorem 4** Let  $A, B \in \mathcal{C}$  such that  $A \triangleleft_O B$ . Then it exists a renaming function  $\phi$  such that  $\phi(A) \blacktriangleleft B$ .

*Proof.(Sketch)* Let  $A = \langle F_A, \text{init}_A, M_A \rangle$  and  $B = \langle F_B, \text{init}_B, M_B \rangle$ . By definition of  $\triangleleft$ , for each  $m_B \in M_B$  there exists a method  $m_A \in M_A$  with the same name. So, as for methods the renaming function is simply the identity function. As for fields, by theorem hypotheses we have that the domain of observables is built on some  $\langle P, \sqsubseteq \rangle \in \mathcal{A}(\mathcal{P}(\Sigma))$ , and that the class invariants for A and B, resp.  $I_A$  and  $I_B$ , are such that  $I_A \sqsubseteq I_B$ . Now, let  $f_A \in F_A$  and let  $\pi \in [F_A \times P \rightarrow P]$  the projection. If  $f_A \in F_B$  then  $\phi(f_A) = f_A$ . If not, we must distinguish two cases. In the first case, there exists  $f' \in F_B. \pi_{f_A}(I_A) \sqsubseteq \pi_{f'}(I_B)$ , so that we can set  $\phi(f_A) = f'$ . Otherwise, there not exists such a  $f'$  in the subclass. By

definition of  $\triangleleft$ , and soundness of projection, this implies that  $f_A$  is not defined in the subclass. The proof concludes by noticing that  $\phi$  must be onto  $F_B$ , because if not the hypothesis  $I_A \sqsubseteq I_B$  is contradicted. *q.e.d.*

## 6 Domain Structure

In general, two classes may have the same interface without being the same class. Analogously, two classes may have the same observable without being the same class. As a consequence, it turns out that both  $\blacktriangleleft$  and  $\triangleleft$  induce a pre-order on  $\mathcal{C}$ . Nevertheless, because of Th. 3 and Th. 4 in general the two pre-orders do not coincide:

**Lemma 5**  $\langle \mathcal{C}, \blacktriangleleft \rangle$  and  $\langle \mathcal{C}, \triangleleft \rangle$  are pre-orders.

Let  $\star$  denote either  $\blacktriangleleft$  or  $\triangleleft$ . The pre-order can be lifted to a partial order by considering the equivalence relation  $\equiv_{[\star]}$  defined as

$$A \equiv_{[\star]} B \iff A \star B \wedge B \star A,$$

and the quotient of  $\mathcal{C}$  w.r.t.  $\equiv_{[\star]}$ , that is  $\mathcal{C}_{/\equiv_{[\star]}}$ . With an abuse of notation, we will not differentiate between an equivalence class (i.e. a set) and a *representative* of the equivalence class (i.e. an element of the set).

**Lemma 6**  $\langle \mathcal{C}_{/\equiv_{[\star]}}, \triangleleft_{/\equiv_{[\star]}} \rangle$  is a partial order.

### 6.1 Bounds

Let  $A$  and  $B$  be classes. By definition of  $\blacktriangleleft$ , the *immediate* syntactic subclasses of  $A$  and  $B$  are all the classes  $C$  such that  $\iota(C) = \iota(A) \cup \iota(B)$ . Dually, the *immediate* syntactic superclasses are such that  $\iota(C) = \iota(A) \cap \iota(B)$ . As a consequence, we have that:

**Theorem 5** Let  $A$  and  $B$  be classes, and let

$$\begin{aligned} A \sqcap_{\blacktriangleleft} B &= \{C \mid \iota(C) = \iota(A) \cup \iota(B)\}, \\ A \sqcup_{\blacktriangleleft} B &= \{C \mid \iota(C) = \iota(A) \cap \iota(B)\}. \end{aligned}$$

Then  $\langle \mathcal{C}_{/\equiv_{[\star]}}, \blacktriangleleft_{/\equiv_{[\star]}} \rangle$  is a partial order with join  $\sqcup_{\blacktriangleleft}$  and meet  $\sqcap_{\blacktriangleleft}$ .

It is worth noting that the syntactic join and the syntactic meet of two classes can be computed easily.

As for semantic subclassing is concerned, the subclasses of  $A$  and  $B$  are all the classes whom observable imply that of both  $A$  and  $B$ . A class  $C$  is an *immediate* subclass if its observable is the greatest one that satisfy such a property. The *immediate* superclass is the dual notion. As a consequence, one may think to define:

$$\begin{aligned} A \sqcap'_{\triangleleft} B &= \{C \mid \bar{C}[C] = \bar{C}[A] \sqcap_o \bar{C}[B]\}, \\ A \sqcup'_{\triangleleft} B &= \{C \mid \bar{C}[C] = \bar{C}[A] \sqcup_o \bar{C}[B]\}. \end{aligned} \quad (7)$$

Nevertheless, in general such definitions are not correct since, for instance, even if  $o = \bar{C}[A] \sqcap_o \bar{C}[B]$  always exists<sup>3</sup> it may be possible that there is no class  $C$  such that  $\bar{C}[C] = o$ . Therefore, additional hypotheses are needed. A first solution is to consider *full-abstract* domains, so that the equalities in (7) are well defined:

**Definition 6 (Full-abstract Domain)** A domain of observables  $\langle O, \sqsubseteq_o \rangle$  is a full-abstract domain if  $\forall o \in O. \exists C \in \mathcal{C}. \bar{C}[C] = o$ .

However, from the previous definition it turns out that the cardinality of  $O$  must be at most that of  $\mathcal{C}$ . This implies that  $O$  contains numerably many elements so that several common-used abstract domains are not full-abstracts.

**Example 7** Domains like the ones for parity and congruences (Ex. 1) are fully abstract, whereas a more sophisticated domain like as Polyhedra is not fully-abstract as it has the cardinality of continuum.

In the case of non fully-abstract domains the optimality requirement has to be dropped, giving rise to the following, weaker, result:

**Lemma 7** Let  $A$  and  $B$  be two classes and  $\text{maxEls}(X)$  the function that returns the maximal elements of  $X$  w.r.t.  $\sqsubseteq_o$ . Then

$$\begin{aligned} A \sqcap_{\triangleleft} B &= \text{maxEls}(\{C \mid \bar{C}[C] \sqsubseteq_o \bar{C}[A] \sqcap_o \bar{C}[B]\}), \\ A \sqcup_{\triangleleft} B &= \text{maxEls}(\{C \mid \bar{C}[C] \sqsupseteq_o \bar{C}[A] \sqcup_o \bar{C}[B]\}). \end{aligned}$$

are such that  $\forall L \in A \sqcap_{\triangleleft} B$  and  $\forall U \in A \sqcup_{\triangleleft} B$ ,  $L \triangleleft A \triangleleft U$  and  $L \triangleleft B \triangleleft U$ .

**Example 8** Let Even and Odd be as in Fig. 1. Then  $\text{Even} \sqcup_{\triangleleft} \text{Odd} = \{C \mid \bar{C}[C] = \langle \mathbb{Z}, \{x = 0\}, \{\text{add}, \text{sub} : (\perp, \mathbb{Z}) \rightarrow \mathbb{Z}\} \rangle\}$ . So,  $\text{Integer} \in \text{Even} \sqcup_{\triangleleft} \text{Odd}$ .

Next lemma states, according to the intuition, that if a class  $A$  is a semantic subclass of a class  $B$  then  $B$  is the most *precise* semantic superclass of the two and  $A$  is the most *precise* semantic subclass of the two.

**Lemma 8** If  $A$  and  $B$  are such that  $A \triangleleft_o B$  then  $A \sqcup_{\triangleleft} B = B$  and  $A \sqcap_{\triangleleft} B = A$ .

### 6.2 Approximation of $\sqcup_{\triangleleft}$

The first refactoring transformation we consider returns a common *semantic* superclass (CSS) of two given classes. By Lemma 7 and Lemma 8 such a superclass always exists. Now we show an algorithm to compute

<sup>3</sup>Recall that from Th. 1 a domain of observables is a complete lattice.

```

CSS(A, B):
  let A = ⟨FA, initA, MAB, initB, MBC = ∅, initC = initA, MC = ∅
  repeat
    select f ∈ FA − FC
    if B ⊲⟨FC ∪ {f}, τFC ∪ {f}(initA), τFC ∪ {f}(MC)⟩
      then FC = FC ∪ {f},
           initC = τFC ∪ {f}(initA)
    || select m ∈ MA − MC
      if B ⊲⟨FC, initC, τFC(MC ∪ {m})⟩
        then MC = MC ∪ {m}
  until no more fields of methods are added

  return ⟨FC, initC, τFC(MC)⟩

```

Figure 5: Algorithm for computing the CSS

it. Such an algorithm is effective as far as the underlying abstract domain of observables is suitable for static analyses, i.e. its elements are computer-representable and it is endowed with a widening operator that guarantees convergence of fixpoint computation [10].

We begin by recalling the definition of meaning-preserving transformation  $\tau$  [14]:

**Definition 7 (Program Transformation)** Let  $A = \langle F, \text{init}, M \rangle$  and  $\langle \alpha, \gamma \rangle$  a pair of functions satisfying (3). A meaning-preservation program transformation  $\tau \in [F \rightarrow M \rightarrow M]$  is such that  $\forall f \in F. \forall m \in M.$

- $\tau_f(m)$  does not contain the field  $f$ ;
- $\forall \bar{d} \in P. \alpha(M^>[m](\gamma(\bar{d}))) \sqsubseteq \alpha(M^>[\tau_f(m)](\gamma(\bar{d})))$ .

Intuitively,  $\tau_f(m)$  projects out the field  $f$  from the source of  $m$  preserving the semantics up to an observation (i.e.  $\alpha$ ).

The algorithm CSS is presented in Fig. 5. It is parameterized by the underlying abstract domain of observables and a meaning preserving map  $\tau$ . The algorithm takes two classes  $A$  and  $B$  as input and it returns a class that is a common superclass to the two. It starts with a superclass for  $A$  (i.e.  $\langle \emptyset, \text{init}_A, \emptyset \rangle$ ). Then, it iterates by non-deterministically adding, at each step, a field or a method of  $A$ : if such an addition produces a superclass for  $B$  then it is retained, otherwise it is discarded. When no more methods or fields can be added, the algorithm returns a semantic superclass for  $A$  and  $B$ , as guaranteed by the following theorem

**Theorem 6 (Soundness of CSS)** Let  $A$  and  $B$  two classes. Then  $CSS(A, B)$  is such that  $A \triangleleft CSS(A, B)$  and  $B \triangleleft CSS(A, B)$ .

*Proof.(Sketch)* We have to prove that  $CSS(A, B)$  is a superclass for the two classes. As for  $A$ , soundness follows from the fact that the algorithm starts with one of its superclass and, by Def. 7 all the *soundly* methods added. As for  $B$ , soundness is checked at each iteration step. *q.e.d.*

It is worth noting that in general,  $CSS(A, B) \neq CSS(B, A)$ . Furthermore, by Th. 1, it follows that if  $\triangleleft$  is decidable, then the algorithm is effective. This is the case when the underlying abstract domain of observables corresponds to one used for a static analysis [21].

**Example 9** Let us consider the classes *MultEight* and *MultTwelve* of the running example and *MultFour* defined as in Sect. 2. Then, when using the abstract domain of linear congruences,  $CSS(\text{MultEight}, \text{MultTwelve}) = \text{MultFour}$ .

## 7 Class Hierarchies

We exploit the results of the previous sections in order to introduce the concept of semantic class hierarchy. Intuitively, a semantic class hierarchy is a tree of classes with the property that each successor node is a semantic subclass of its parent. Then, we use the notion of semantic class hierarchy to define and prove correct a set of operators for the manipulation and the transformation of class hierarchies.

### 7.1 Semantic Class Hierarchies

In the setting of this paper, we consider just single inheritance. As a consequence, class hierarchies have the form of a tree.

**Definition 8** Let  $T$  be a tree. Then  $\text{nodesOf}(T)$  denotes the elements of the tree,  $\text{rootOf}(T)$  denotes the root of the tree, and if  $n \in \text{nodesOf}(T)$  then  $\text{sonsOf}(n)$  are the successors of the node  $n$ . In particular, if  $\text{sonsOf}(n) = \emptyset$  then  $n$  is a leaf. A tree with a root  $r$  and successors  $S$  is  $\text{tree}(r, S)$ .

Intuitively, a hierarchy admissible w.r.t. a transitive relation  $\rho$  on classes is such that all the nodes of the tree are classes, and given two nodes  $n$  and  $n'$  such that  $n' \in \text{sonsOf}(n)$  then  $n'$  is in the relation  $\rho$  with  $n$ . Formally:

**Definition 9 (Admissible Class Hierarchy)** Let  $\mathcal{H}$  be a tree and  $\rho \subseteq \mathcal{C} \times \mathcal{C}$  be a transitive relation on classes. Then we say that  $\mathcal{H}$  is class hierarchy admissible w.r.t.  $\rho$ , if

- $\text{nodesOf}(\mathcal{H}) \subseteq \mathcal{C}$

$$- \forall n \in \text{nodesOf}(\mathcal{H}). \forall n' \in \text{sonsOf}(n). n' \rho n.$$

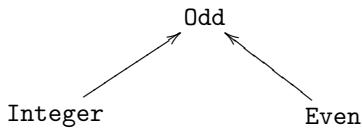
We denote the set of all the class hierarchies admissible w.r.t.  $\rho$  as  $\mathbb{H}[\rho]$ . It is worth noting that our definition subsumes the definition of class hierarchies present in mainstream object-oriented languages. In fact, when  $\rho$  is instantiated with  $\ll$ , we obtain class hierarchies in which all the subclasses have at least the same methods that their superclass.

A semantic class hierarchy is just the instantiation of Def. 9 with the relation  $\triangleleft$ :

**Definition 10 (Semantic Hierarchy)** *A class hierarchy admissible w.r.t.  $\triangleleft$  is a semantic class hierarchy.*

Note that as  $\triangleleft$  is trivially a transitive relation, the definition above is well-stated. With an abuse of language, in the following we say that a class hierarchy is admissible if it is a semantic hierarchy.

**Example 10** *Let us consider the classes Odd, Integer and Even of our running example. Then the following class hierarchy,  $\mathcal{G}$ ,*



*is admissible for  $\ll$ . Nevertheless,  $\mathcal{G}$  is not admissible for  $\triangleleft_{\text{Parity}}$ .*

## 7.2 Order on Class Hierarchies

A pre-order can be defined on admissible class hierarchies. It captures the intuition that a class hierarchy  $\mathcal{H}_1$  is more precise than  $\mathcal{H}_2$  if it contains *at least* all the information of  $\mathcal{H}_2$ .

**Definition 11 (Order on Hierarchies)** *Let  $A \triangleleft^* B \in \mathcal{H}$  denote the fact that there is a path from  $B$  to  $A$  in the admissible hierarchy  $\mathcal{H}$  and let  $\mathcal{H}_1$  and  $\mathcal{H}_2$  be two admissible class hierarchies. Then the order relation  $\ll$  is defined as*

$$\begin{aligned} \mathcal{H}_1 \ll \mathcal{H}_2 &\iff \forall A \triangleleft^* B \in \mathcal{H}_2. \\ &\exists A' \triangleleft^* B' \in \mathcal{H}_1. A' = A \wedge B' = B. \end{aligned}$$

**Lemma 9** *Let  $\mathcal{H}_1$  and  $\mathcal{H}_2$  two class hierarchies such that  $\mathcal{H}_1 \ll \mathcal{H}_2$ . Then  $\text{nodesOf}(\mathcal{H}_2) \subseteq \text{nodesOf}(\mathcal{H}_1)$ .*

**Example 11** *Let us consider the two class hierarchies in Fig. 6. When considering parities the two are admissible. We have that  $\mathcal{K}_1 \ll \mathcal{K}_2$ . On the other hand, as Even does not belong to  $\mathcal{K}_2$ , by Lemma 9,  $\mathcal{K}_2 \not\ll \mathcal{K}_1$ .*

```

 $\mathcal{H} \uplus C =$ 
  let  $R = \text{rootOf}(\mathcal{H})$ ,  $S = \text{sonsOf}(R)$ 
  let  $\mathcal{H}_< = \{\mathcal{K} \in S \mid \text{rootOf}(\mathcal{K}) \triangleleft C\}$ 
  let  $\mathcal{H}_> = \{\mathcal{K} \in S \mid \text{rootOf}(\mathcal{K}) \triangleright C\}$ 
  if  $C \in \text{nodesOf}(\mathcal{H})$  then return  $\mathcal{H}$ 
  if  $R \triangleleft C$  then return tree( $C$ ,  $\mathcal{H}$ )
  if  $R \triangleright C$  then
    if  $\mathcal{H}_< \neq \emptyset$  then
      return tree( $R$ ,  $(S - \mathcal{H}_<) \cup \text{tree}(C, \mathcal{H}_<)$ )
    if  $\mathcal{H}_< \neq \emptyset$  then select  $\mathcal{K} \in S$ 
      return tree( $R$ ,  $(S - \mathcal{K}) \cup (C \uplus \mathcal{K})$ )
    else return tree( $R$ ,  $S \cup C$ )
  else select  $C_T = \text{CSS}(R, C)$ 
  return tree( $C_T$ ,  $\{\mathcal{H}, C\}$ )
  
```

Figure 7: The algorithm for a fair class insertion

## 8 Semantic Class Hierarchy Transformations

A *fair* operator on class hierarchies transform a set of class hierarchies admissible w.r.t. a relation  $\rho$  into a class hierarchy which is admissible w.r.t. a relation  $\rho'$ . Formally:

**Definition 12 (Fair Operator)** *Let  $\rho$  and  $\rho'$  be transitive relations. Then we say that a function  $t$  is a fair operator w.r.t.  $\rho$  and  $\rho'$  if  $t \in [\mathcal{P}(\mathbb{H}[\rho]) \rightarrow \mathbb{H}[\rho']]$ .*

Next, we define and prove correct some fair operator. When not differently stated, in the definition above we assume that  $\rho' = \triangleleft$ .

### 8.1 Class Insertion

The first operator we consider is the one for adding a class into an admissible class hierarchy. The algorithm definition of such an operator is presented in Fig. 7. It takes as inputs an admissible class hierarchy  $\mathcal{H}$  and a class  $C$ . If  $C$  already belongs to  $\mathcal{H}$  then the hierarchy is unchanged. Otherwise, if  $C$  is a superclass of the root class of  $\mathcal{H}$  then a new class hierarchy whom root is  $C$  is returned. On the other hand, if  $C$  is a subclass of the root class of  $\mathcal{H}$  then it must be inserted somewhere in the successors of the root so to preserve the *admissibility* of the hierarchy. If  $C$  is a superclass for some of the successors, then it is inserted between the root of  $\mathcal{H}$  and such successors. Otherwise we see whether some root class of the successors is a superclass of  $C$ . If it is the case, then the algorithm is recursively applied, otherwise  $C$  is added at this level of the hierarchy. Finally, if  $C$  and the root of  $\mathcal{H}$  are unrelated, the algorithm returns a new hierarchy whom root is a superclass of both  $C$  and the root of  $\mathcal{H}$ .

The soundness of the algorithm, and that of the algorithms we will present later, is a consequence of the

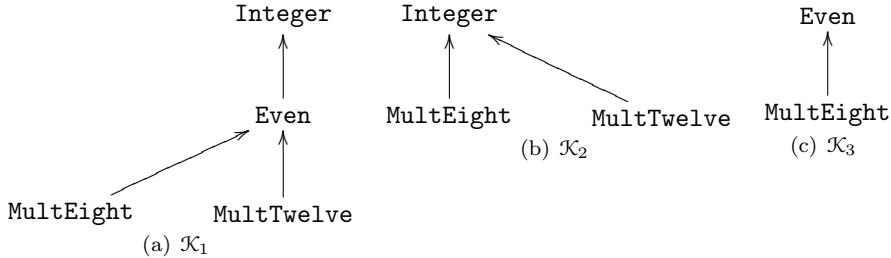


Figure 6: Example Hierarchies such that  $\mathcal{K}_1 \ll \mathcal{K}_2$  and  $\mathcal{K}_1 \ll \mathcal{K}_3$

following lemma which essentially states that if there is a path from a class  $B$  to a class  $A$  in an admissible hierarchy then there is also a path in the extended hierarchy:

**Lemma 10** *Let  $\mathcal{H} \in \mathbb{H}[\triangleleft]$  and  $C \in \mathcal{C}$ . Then  $\sqcup$  is such that  $\forall \mathcal{H} \in \mathbb{H}[\triangleleft]. \forall C. \mathcal{H} \sqcup C \ll \mathcal{H}$ .*

**Lemma 11 (Class Insertion)** *The operator  $\sqcup$  defined in Fig. 7 is a fair operator w.r.t.  $\triangleleft$ , i.e.  $\sqcup \in [\mathbb{H}[\triangleleft] \times \mathcal{C} \rightarrow \mathbb{H}[\triangleleft]]$ .*

**Example 12** *Let us consider the hierarchies of Ex. 11. Then  $\mathcal{K}_2 \sqcup \text{Even} = \mathcal{K}_1$  and  $\mathcal{K}_1 \sqcup \text{Even} = \mathcal{K}_1$*

It is worth noting that once again  $\sqcup$  is effective as soon as the underlying domain of observables is suitable for a static analysis.

The dual operator of class insertion, i.e. the elimination of a class from a hierarchy, corresponds straightforwardly to the algorithm for removing a node from an ordered tree [7].

## 8.2 Hierarchy Building

Now we address the question of building an admissible class hierarchy starting from a syntactic class hierarchy. The operator for performing such an action,  $\beta$ , is built on the top of  $\sqcup$ . Roughly,  $\beta(\mathcal{S})$  creates a new, admissible, hierarchy by systematically inserting all the nodes of  $\mathcal{S}$ :

```

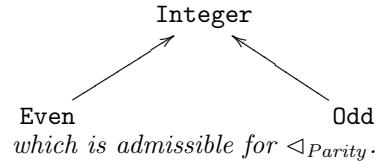
 $\beta(\mathcal{S}) =$ 
  let  $\mathcal{H} = \emptyset, N = \text{nodesOf}(\mathcal{S})$ 
  while  $N \neq \emptyset$  do
    select  $C \in N$ 
     $\mathcal{H} = \mathcal{H} \sqcup C, N = N - C$ 
  return  $\mathcal{H}$ 

```

The soundness of  $\beta$  is stated by the next lemma. Intuitively, when applied to a syntactic class hierarchy  $\mathcal{S}$ ,  $\beta(\mathcal{S})$  returns a semantic class hierarchy whom nodes are those of  $\mathcal{S}$ .

**Lemma 12** *The algorithm  $\beta$  is a fair operator w.r.t.  $\blacktriangleleft$  and  $\triangleleft$ , i.e.  $\beta \in [\mathbb{H}[\blacktriangleleft] \rightarrow \mathbb{H}[\triangleleft]]$ .*

**Example 13** *Let us consider the hierarchy  $\mathcal{G}$  of Ex. 10. Then  $\beta(\mathcal{G})$  is*



which is admissible for  $\triangleleft_{\text{Parity}}$ .

## 8.3 Merging of Hierarchies

As a last example of instantiation of our framework, we consider the operator for merging two admissible hierarchies. Intuitively, such an operator return an admissible class hierarchy which contains all the classes of the input hierarchies. Formally, it is defined as

```

 $\mathcal{H}_1 \sqcup \mathcal{H}_2 =$ 
  let  $\mathcal{H} = \mathcal{H}_1, N = \text{nodesOf}(\mathcal{H}_2)$ 
  while  $N \neq \emptyset$  do
    select  $C \in N$ 
     $\mathcal{H} = \mathcal{H} \sqcup C, N = N - C$ 
  return  $\mathcal{H}$ .

```

**Lemma 13**  *$\sqcup$  is a fair operator w.r.t.  $\triangleleft$ , i.e.  $\sqcup \in [\mathbb{H}[\triangleleft] \rightarrow \mathbb{H}[\triangleleft]]$ . Furthermore,  $\forall \mathcal{H}_1, \mathcal{H}_2. \mathcal{H}_1 \sqcup \mathcal{H}_2 \ll \mathcal{H}_1$ .*

**Example 14** *Consider the hierarchies  $\mathcal{K}_2$  and  $\mathcal{K}_3$  of Fig. 6. Then  $\mathcal{K}_2 \sqcup \mathcal{K}_3 = \mathcal{K}_1$ .*

## 9 Related Work

The notion of inheritance first appeared in Simula 67 [16]. In Simula subclasses inherit all the attributes of their superclasses. Smalltalk [17] adopts and exploits the idea of inheritance, in particular by stressing the message-passing paradigm. Furthermore, with respect to Simula, Smalltalk abandons static scoping and strong typing in order to gain flexibility and to implement system introspection. Finally, Simula and Smalltalk both

allow *multiple* inheritance, i.e. a class may have several incomparable superclasses. The inheritance support in C++ [31] and Object Pascal [1] is very close to that of Smalltalk. Last generation object-oriented languages as Java [18] and C# [24] provide a form of inheritance through class extension and interface implementation. Roughly, they have a weaker concept of multiple inheritance: a class has exactly one superclass but it may *implement* an arbitrarily number of interfaces. An interface is a class without the implementation of the methods.

In their seminal work on Simula [16], Dahl and Nygaard justified the concept of inheritance on syntactic bases, namely as textual concatenation of program blocks. A first semantic approach is [17] where the authors introduced an (informal) operational approach to the semantics of inheritance. In particular they reduced the problem of specifying the semantics of message dispatch to that of method lookup. In [6] a denotational characterization of inheritance is introduced and proved correct w.r.t. to an operational semantics based on the method lookup algorithm of [17]. An unifying view of the different forms of inheritance provided by programming languages is presented in [2]. The authors present an inheritance mechanism, based on composition of mixins, that subsumes the others.

In the *objects as records model* [3], the semantics of an object is abstracted with its type so that the inheritance is identified with subtyping and the semantics of inheritance boils down to the subtyping relation. Nevertheless, such an approach is not fully satisfactory as shown in [5]. The notion of subtyping has been generalized in [20] where the authors introduced the view of inheritance as property preservation: the *behavioral type* of a class is a human-provided formula, which specifies the behavior of the class, and subclassing boils down to formula implication. In such a view a class is a subclass of another if its behavioral type implies the one of the other. The checking of the implication can be done by hand or using a theorem prover. The main differences between our concept of observable and that of behavioral type is that observables are systematically obtained as an abstraction of the class semantics instead of being provided by the programmer. Furthermore, the checking of semantic subclassing,  $\triangleleft$ , is automatic when the underlying abstract domain corresponds to one for a static analysis.

As for class hierarchies refactoring, [29] presents a semantic-preserving approach to class composition. Such an approach preserves the behavior of the composing hierarchies when they do not interfere. If this is the case, a static analysis determines which components (classes, methods, etc.) of the hierarchies may interfere, given a set of programs that use such hierar-

chies. Such an approach is the base of the [30], which exploits static and dynamic information for class refactoring. The main difference between these works and ours is that we exploit the notion of observable, which is a property valid for *all* the instantiation contexts of a class. As a consequence we do not need to rely on a set of test programs for inferring hierarchy properties. Furthermore, as soundness requirement, we ask that a refactoring operator on a class hierarchy preserve the observable, i.e. an abstraction of the concrete semantics. As a consequence we are in a more general setting, and the traditional one is recovered as soon as we consider as domain of observables the concrete one.

## 10 Conclusions and Future Work

We introduced a framework for the definition and the manipulation of class hierarchies based on the concept of observable as abstraction of the semantics. This approach is built on top of abstract interpretation theory, a general technique introduced in [10] that has already been successfully applied to different areas (as automatic program verification, program optimization, semantics, etc.).

As for future work, we plan to extend the set of operators on class hierarchies and to study their algebraic properties. Furthermore, we plan to consider observables extracted from programs that use a given class hierarchy, e.g. so to capture the part of the class hierarchy that is effectively used by a set of programs.

## References

- [1] Borland Inc. *Turbo Pascal 5.5 Object Oriented Programming Guide*. Borland Inc., 1989. Available online at [http://community.borland.com/article/images/20803/TP\\_55\\_OOP\\_Guide.pdf](http://community.borland.com/article/images/20803/TP_55_OOP_Guide.pdf).
- [2] G. Bracha and W. R. Cook. Mixin-based inheritance. In *5th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '90)*, volume 25(10) of *SIGPLAN Notices*, pages 303–311, October 1990.
- [3] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67, Berlin, 1984. Springer-Verlag. Full version in *Information and Computation*, 76(2/3):138–164, 1988.
- [4] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.
- [5] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th annual ACM*

- SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'90)*. ACM Press, January 1990.
- [6] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, November 1994.
  - [7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
  - [8] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, September 1977.
  - [9] P. Cousot. Types as abstract interpretations, invited paper. In *24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 316–331. ACM Press, January 1997.
  - [10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, January 1977.
  - [11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.
  - [12] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, July 1992.
  - [13] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
  - [14] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 178–190. ACM Press, New York, NY, January 2002.
  - [15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM Press, 1978.
  - [16] O. Dahl and K. Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM (CACM)*, 9(9):671–678, September 1966.
  - [17] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
  - [18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification - 2nd Edition*. Sun Microsystems, 2001.
  - [19] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 464 of *Lectures Notes in Computer Science*, pages 169–192. Springer-Verlag, April 1991.
  - [20] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
  - [21] F. Logozzo. An approach to behavioral subtyping based on static analysis. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, April 2004.
  - [22] F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lectures Notes in Computer Science*, pages 211–222. Springer-Verlag, January 2004.
  - [23] B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.
  - [24] Microsoft Inc. *Microsoft C# Language Specifications*. Microsoft Press, 2001.
  - [25] NetBeans.org and Sun Mycosystem, Inc. Netbeans IDE, 2004. <http://www.netbeans.org/>.
  - [26] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.
  - [27] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of Java programs. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP '01)*, volume 2072 of *Lectures Notes in Computer Science*, pages 77–98. Springer-Verlag, 2001.
  - [28] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 1–13. ACM Press, January 2004.
  - [29] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *Lectures Notes in Computer Science*, pages 562–584. Springer-Verlag, June 2002.
  - [30] M. Streichenbach and G. Snelting. Refactoring class hierarchies with KABA. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, 2004.
  - [31] B. Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2000.