

# Semantic Hierarchy Refactoring by Abstract Interpretation

Francesco Logozzo<sup>1</sup> and Agostino Cortesi<sup>2</sup>

<sup>1</sup> École Normale Supérieure, France

<sup>2</sup> Università Ca' Foscari di Venezia, Italy

**Abstract.** A semantics-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 focusing on a behavioral property of interest. We define a 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. The approach is then extended to class hierarchies, leading to a semantics-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.

## 1 Introduction

In the object-oriented paradigm, a crucial role is played by the notion of class hierarchy. Being  $A$  a subclass of  $B$  captures the fact that the state and the behavior of the elements of  $A$  are coherent with the intended meaning of  $B$ , while disregarding the additional features and functionalities that characterize the subclass.

The approach of mainstream object-oriented languages, like Java and C++, to class hierarchies can be seen as merely syntactic. In such a view hierarchies are collections of classes ordered by the transitive closure of explicitly declared subclass or subtype relations. This is why the main theoretical and practical contributions to hierarchy refactoring issues [32, 33] combine static and dynamic analyses that focus only on syntactic elements. However, as pointed out by [29], this approach has severe limitations, as it leads to troubles when trying to face the issue of extending a given class hierarchy.

In this paper we adopt an alternative, semantics-based approach for the definition and manipulation of class hierarchies. It uses previous works on abstract interpretation theory [10], that allows formalizing the notion of different levels of property abstraction and of abstract semantics. This framework is based on the notion of observable of a class, i.e., an abstraction of the class semantics that focuses on a behavioral property of interest. The intuition 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 these observables. 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 become even more interesting when facing the problem of manipulating class hierarchies which has more than thousands of classes (for instance, NetBeans [27] is made up of 8328 classes). We formalize the notion of semantic ordering of hierarchies as “*when is it the case that a hierarchy is more informative with respect to a given observable?*”

We show that this notion of semantic subclassing

- can be formally related to the traditional syntactic-based subclassing relation;
- it is crucial for designing automatic and modular verification tools for polymorphic code;
- it enlightens the trade-off between the expressive power of specification languages for object-oriented languages and the subtype relations they support;
- it is the base to design algorithms and tools for extending, refactoring and merging class hierarchies.

In fact, in the paper we show how it can be used for the automatic and modular verification of polymorphic code, for bounding the expressive power of specification languages for object-oriented languages and for the characterization of semantic class hierarchies. Intuitively, semantic class hierarchies ensure that, up to a given observable, classes lower in the hierarchy specializes the behavior of the upper classes. We instantiate our framework by design algorithms for extending, refactoring and merging class hierarchies. Such algorithms represent the basis for our mid-term goal, that is a tool for the modular verification and the semi-automatic refactoring of large class hierarchies.

**Paper Structure.** In Section 2, an example introduces the main ideas of the paper. In Section 3, the notion of observable is introduced as an abstraction of the concrete semantics. In Section 4, we introduce the semantic subclass relation, we discuss its relationship with the syntactic notion, and we show its use for modular verification of polymorphic code. In Section 5, the framework is lifted to class hierarchies by introducing a suite of refactoring operators. Finally, Section 6 discusses related work, and Section 7 concludes.

## 2 A Motivating Example

Let us consider the five classes described in Fig. 1 that encode different sets of integer numbers. In class **Even**, variable **x** can only take even values, whereas variable **x**

```

class Integer {
  int x;
  init(){ x = 0 }
  add() { x += 1 }
  sub() { x -= 1 } }

class Even {
  int x;
  init(){ x = 0 }
  add() { x += 2 }
  sub() { x -= 2 } }

class Odd {
  int x;
  init(){ x = 1 }
  add() { x += 2 }
  sub() { x -= 2 } }

class MultEight{
  int x;
  init(){ x = 0 }
  add() { x += 16 }
  sub() { x -= 8 } }

class MultTwelve{
  int x;
  init(){ x = 0 }
  add() { x += 24 }
  sub() { x -= 12 } }

```

Fig. 1. Running examples

of `Odd` takes odd values only. The instance variable of `MultEight` and `MultTwelve` can only be assigned a value that is a multiple of 8 and 12, respectively.

A first question to address is “*What are the admissible hierarchies among such classes?*”. A hierarchy is admissible when the subclasses preserve a given property of their superclass. So, when the *parity* of the field `x` is observed, both the class hierarchies  $\mathcal{H}_1$  and  $\mathcal{H}_2$  in Fig. 2 are admissible. This is true for  $\mathcal{H}_1$ , as the value of `MultEight.x` is always a multiple of 8, and in particular it is even. As a consequence, when just parity is observed, `MultEight` preserves the behavior of `Even`. On the other hand,  $\mathcal{H}_2$  is also an admissible class hierarchy w.r.t. parity as the values taken by `MultTwelve.x` and `MultEight.x` are even numbers, too. As a consequence, `MultTwelve` preserves the parity behavior of its superclass `MultEight`. Nevertheless, when we consider a more precise property, for instance the value taken by `x` up to a numerical *congruence*, then  $\mathcal{H}_2$  is no longer an admissible hierarchy. In fact, as in general a multiple of 12 is not a multiple of 8, `MultTwelve` does not preserve the congruence property of its ancestor `MultEight`.

“*Why do we need admissible class hierarchies?*” For two reasons: (i) it allows one to design modular verification tools of polymorphic methods, and (ii) it supports design of semantics-preserving operations on class hierarchies.

To illustrate (i), consider the class hierarchy  $\mathcal{H}_1$  and the method `inv`, defined as follows:

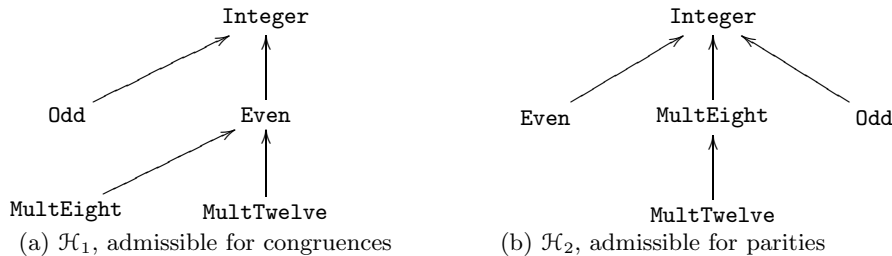


Fig. 2.  $\mathcal{H}_1$  and  $\mathcal{H}_2$ , two possible class hierarchies

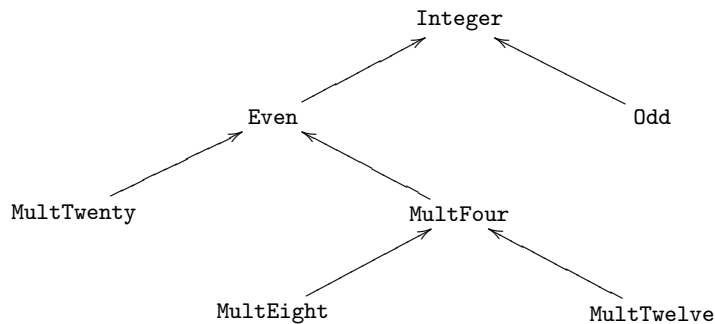
```
inv(Even e){return 1/(1 - e.x%2)}.
```

In order to prove that `inv` never performs a division by zero, it suffices to prove it w.r.t. `Even` instances. In fact as  $\mathcal{H}_1$  is admissible for parity, then all the subclasses of `Even` preserve the property that `x` is an even number. Nevertheless, in order to prove it correct also for all the future extensions of the hierarchy, we need to assure that all the manipulations on class hierarchies preserve its admissibility. This leads to (ii).

This semantic approach can be used to define, and prove correct, manipulating operations on class hierarchies that preserve admissibility w.r.t. a given property. For instance, we will show an algorithm for class insertion. Such an algorithm, when applied to the classes of Fig. 3 and to the hierarchy  $\mathcal{H}_1$ , returns the hierarchy  $\mathcal{H}_3$  in Fig. 4, which is still admissible for congruences (and hence parities). As a consequence, the method `inv` is still guaranteed to be correct for all possible inputs.

```
class MultFour {      class MultTwenty {
  int x;              int x;
  init() { x = 0 }   init() { x = 0 }
  add()  { x += 4 }   add()  { x += 20 }
  sub()  { x -= 4 }}  sub()  { x -= 60 }}
```

**Fig. 3.** Two classes to be added to  $\mathcal{H}_1$



**Fig. 4.**  $\mathcal{H}_3$ : the hierarchy  $\mathcal{H}_1$  augmented with `MultTwenty` and `MultFour`

### 3 Concrete and Abstract Semantics of Classes

In this section, we introduce the syntax and the concrete semantics of classes. Then, we define the domain of observables and the abstract semantics of a class.

#### 3.1 Syntax

A class is a template for objects. It is provided by the programmer who specifies the fields, the methods and the class constructor.

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

Like in Smalltalk [15], methods are untyped and fields are private. This is just to simplify the exposition and it does not cause any loss of generality: any external access to a field  $\mathbf{f}$  can be simulated by a pair of methods  $\mathbf{set\_f}/\mathbf{get\_f}$ . Furthermore, we assume that a class has only one constructor. The generalization to an arbitrary number of constructors is straightforward. The interface of a class is the set of messages it can answer:

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

### 3.2 Concrete Semantics

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

When a class is instantiated, the class constructor is called to set the internal state of the new object. This is modeled by a semantic function  $\mathbf{i}[\mathbf{init}] \in [D_{\text{val}} \rightarrow \mathcal{P}(\Sigma)]$ . We consider sets in order to model non-determinism, e.g., user input or random choices.

The semantics of a method  $\mathbf{m}$  is a function  $\mathbf{m}[\mathbf{m}] \in [D_{\text{val}} \times \Sigma \rightarrow \mathcal{P}(D_{\text{val}} \times \Sigma)]$ . 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  $\langle \text{return value (if any), new object state} \rangle$ .

The most precise state-based property of a class  $\mathbf{C}$  is the set of states reached by any execution of every instance of  $\mathbf{C}$  in any possible context. In this paper, we consider just state-based properties. Such an approach can be shown to be an abstraction of a trace-based semantics for object-oriented languages, [23, 22], in which just the states before and after the invocation of a method are retained.

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

$$S_0 = \{\sigma \in \Sigma \mid \exists v \in D_{\text{val}}. \sigma \in \mathbf{i}[\mathbf{init}](v)\}.$$

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

$$\mathbb{M}^>[\mathbf{m}](S) = \{\sigma' \in \Sigma \mid \exists \sigma \in S. \exists v \in D_{\text{val}}. \exists v' \in D_{\text{val}}. \langle v', \sigma' \rangle \in \mathbf{m}[\mathbf{m}](v, \sigma)\}.$$

The class reachable states are the least solution of the following recursive equations:

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

The above equations characterize the set of states that are reachable before and after the invocation of any method in any instance of the class. Stated otherwise, they consider all the states reached after any possible invocation, in any order, with any input values of the methods of a class. A more general situation, in which the context may update the fields of an object, e.g. , because of aliasing, is considered in [22].

The least solution of (1) w.r.t. set inclusion corresponds to a tuple  $\langle S, S_0, \{m : S_m\} \rangle$  such that  $S$  is a class invariant [21, 23, 22], and for each method  $m$ ,  $S_m$  is the strongest postcondition of the method. The method preconditions can be obtained by going backward from the postconditions: 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 of  $m$ . Formally, the collecting *backward* method semantics  $\mathbb{M}^{\leftarrow}[\![m]\!] \in [\mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)]$  is defined as

$$\mathbb{M}^{\leftarrow}[\![m]\!](S) = \{\sigma \in \Sigma \mid \exists \sigma' \in S. \exists v \in D_{val}. \exists v' \in D_{val}. \langle v', \sigma' \rangle \in m[\![m]\!]\langle v, \sigma \rangle\}.$$

and the methods preconditions are  $B_m = \mathbb{M}^{\leftarrow}[\![m]\!](S_m)$ .

The concrete class semantics, i.e., the most precise property of a class [10], is the triple  $\mathbb{C}[\![C]\!] = \langle S, S_0, \{m : B_m \rightarrow S_m\} \rangle$ .

The use of the concrete semantics  $\mathbb{C}[\![C]\!]$  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, this approach is not suitable for an effective definition of semantic subclassing. Second, it is too precise, as it may differentiate classes that do not need to be 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 `push` and `pop` methods. 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 respective `pop` methods: both of them return the value on the top of the stack.

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

### 3.3 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  $C$ . We 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 is a complete lattice, related to the concrete domain by a Galois connection [10]:

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \stackrel{\gamma}{\longleftarrow} \langle P, \sqsubseteq, \perp, \top, \sqcup, \sqcap \rangle. \quad (2)$$

For instance, if we are interested in the linear relations between the values of the fields of the instances of  $\mathcal{C}$ , we instantiate  $P$  with the Octagons abstract domain [26]. On the other hand if we are interested in object aliasing then we are likely to choose for  $P$  an abstract domain that captures shapes, e.g., [30, 31].

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

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

Intuitively, an element of  $O[P]$  consists of an approximation of the class invariant, the constructor postcondition, and for each method an approximation of its precondition and postcondition. A method precondition is in turn made up of two parts, one for the method input values and the other for that internal object state. When no ambiguity arises, we write  $\langle O, \sqsubseteq_o \rangle$  instead of  $\langle O[P], \sqsubseteq_o^{[P]} \rangle$ . We tacitly assume that if a method  $\mathbf{n}$  is not defined in a class, then its precondition and postconditions are respectively  $\top$  and  $\perp$ .

The partial order  $\sqsubseteq_o$  on  $O$  is 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<sup>1</sup> of  $O$ . Then the order  $\sqsubseteq_o$  is defined as:

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

If  $\mathbf{o}_1$  and  $\mathbf{o}_2$  are the observables of two classes  $\mathbf{A}$  and  $\mathbf{B}$  then the order  $\sqsubseteq_o$  ensures that  $\mathbf{A}$  preserves the class invariant of  $\mathbf{B}$  and that the methods of  $\mathbf{A}$  are a “safe” replacement of those with the same name in  $\mathbf{B}$ . Intuitively, the precondition generalizes the observations, made in the context of type theory, of [3]. 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.,  $\bar{Q}_i \sqsubseteq \bar{R}_i$ ). Finally, the postcondition of the inherited method must be at least as strong as that of the ancestor (i.e.,  $\bar{I}_i \sqsubseteq \bar{J}_i$ ).

Having defined  $\sqsubseteq_o$ , it is routine to check that  $\perp_o = \langle \perp, \perp, \{ \mathbf{m}_i : \langle \top, \top \rangle \rightarrow \perp \} \rangle$  is the smallest element of  $O$  and  $\top_o = \langle \top, \top, \{ \mathbf{m}_i : \langle \perp, \perp \rangle \rightarrow \top \} \rangle$  is the largest one. The join,  $\sqcup_o$ , and the meet,  $\sqcap_o$ , operators on  $O$  can be defined point-wise.

Suppose that the order relation  $\sqsubseteq$  on  $P$  is decidable [28]. This is the case for abstract domains used for effective static analyses. 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.

**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.*

From basic abstract interpretation theory [11] we know that  $\mathcal{A}(\mathcal{P}(\Sigma))$ , the set of all the abstractions of the concrete domain, is a complete lattice ordered w.r.t.

<sup>1</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  $\mathbf{o}_2$ .

the “relative” precision,  $\leq$ , of abstract domains. As immediate consequence, we obtain that Galois connections can be lifted to the domain of observables:

**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,*

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

where  $\alpha_o$  and  $\gamma_o$  are

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

### 3.4 Abstract Semantics

Once the abstract domain is defined, an abstraction of  $\mathbb{C}[\mathbb{C}]$  can be obtained by considering the abstract counterpart for (1). As a first step we need to consider the abstraction corresponding to the initial states, and the forward and the backward collecting semantics. We consider the *best* abstract counterparts for such concrete semantic functions.

By Galois connection properties, the best approximation for the initial states of the class is  $\alpha(S_0) = \bar{S}_0$ . By [11], the best approximation in  $P$  of the forward collecting method semantics of  $\mathbf{m}$  of  $\mathbb{C}$  is  $\bar{\mathbb{M}}^>[\mathbf{m}] \in [P \rightarrow P]$  defined as  $\bar{\mathbb{M}}^>[\mathbf{m}](\bar{S}) = \alpha \circ \mathbb{M}^>[\mathbf{m}] \circ \gamma(\bar{S})$ . The abstract counterpart for the equations (1) is the following equation system:

$$\begin{aligned} \bar{S} &= \bar{S}_0 \sqcup \bigsqcup_{\mathbf{m} \in \mathbf{M}} \bar{S}_m \\ \bar{S}_m &= \bar{\mathbb{M}}^>[\mathbf{m}](\bar{S}) \quad \mathbf{m} \in \mathbf{M}. \end{aligned} \tag{3}$$

The above equations are monotonic and, by the Tarski fixpoint theorem, there exists a least solution  $\langle \bar{S}, \bar{S}_0, \{\mathbf{m} : \bar{S}_m \} \rangle$ . Similarly to the concrete case, the abstract preconditions can be obtained by considering the best approximation of the backward collecting method semantics  $\bar{\mathbb{M}}^<[\mathbf{m}] \in [P \rightarrow P]$  defined as  $\bar{\mathbb{M}}^<[\mathbf{m}](\bar{S}) = \alpha \circ \mathbb{M}^<[\mathbf{m}] \circ \gamma(\bar{S})$ . The method abstract preconditions are obtained by projecting  $\bar{\mathbb{M}}^<[\mathbf{m}](\bar{S}_m)$  respectively on the method input values and the instance fields:  $\bar{V}_m = \pi_{in}(\bar{\mathbb{M}}^<[\mathbf{m}](\bar{S}_m))$  and  $\bar{B}_m = \pi_F(\bar{\mathbb{M}}^<[\mathbf{m}](\bar{S}_m))$ .

To sum up, the triple  $\mathbb{C}[\mathbb{C}] = \langle \bar{S}, \bar{S}_0, \{\mathbf{m} : \langle \bar{V}_m, \bar{B}_m \rangle \rightarrow \bar{S}_m \} \rangle$  belongs to the domain of observables, and it is the best sound approximation of the semantics of  $\mathbb{C}$ , w.r.t the properties encoded by the abstract domain  $\langle P, \sqsubseteq \rangle$ .

**Theorem 2 (Observable of a Class).** *Let  $\langle P, \sqsubseteq \rangle$  be an abstract domain that satisfies (2) and let the observable of a class  $\mathbb{C}$  w.r.t. the property encoded by  $\langle P, \sqsubseteq \rangle$  be  $\mathbb{C}[\mathbb{C}] = \langle \bar{S}, \bar{S}_0, \{\mathbf{m} : \langle \bar{V}_m, \bar{B}_m \rangle \rightarrow \bar{S}_m \} \rangle$ . Then  $\alpha_o(\mathbb{C}[\mathbb{C}]) \sqsubseteq_o \bar{\mathbb{C}}[\mathbb{C}]$ .*

*Example 1.* Let us instantiate  $\langle P, \sqsubseteq \rangle$  with  $\text{Con}$ , the abstract domain of equalities of linear congruences, [16]. The elements of such a domain have the form

$x = a \bmod b$ , where  $x$  is a program variable and  $a$  and  $b$  are integers. The representation function  $\gamma_c \in [\text{Con} \rightarrow \mathcal{P}(\Sigma)]$  is defined as

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

Let us consider the classes `Even` and `MultiEight` in Fig. 2 and let  $e$  be the property  $x = 0 \bmod 2$ ,  $d$  the property  $x = 1 \bmod 2$  and  $u$  be the property  $x = 0 \bmod 8$ . Then the observables of `Even` and `MultiEight` w.r.t. `Con` are

$$\begin{aligned} \bar{\mathbb{C}}[\text{Even}] &= \langle e, e, \{\text{add} : \langle \perp, e \rangle \rightarrow e, \text{sub} : \langle \perp, e \rangle \rightarrow e\} \rangle \\ \bar{\mathbb{C}}[\text{Odd}] &= \langle d, d, \{\text{add} : \langle \perp, d \rangle \rightarrow d, \text{sub} : \langle \perp, d \rangle \rightarrow d\} \rangle \\ \bar{\mathbb{C}}[\text{MultiEight}] &= \langle u, u, \{\text{add} : \langle \perp, u \rangle \rightarrow u, \text{sub} : \langle \perp, u \rangle \rightarrow u\} \rangle. \end{aligned}$$

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

## 4 Subclassing

The notion of subclassing can be defined both at semantic and syntactic level. Given two classes `A` and `B`, `A` is a *syntactic* subclass of `B`, denoted  $\mathbf{A} \blacktriangleleft \mathbf{B}$ , if all the names defined in `B` are defined in `A` too. On the other hand, `A` is a *semantic* subclass of `B`, denoted  $\mathbf{A} \triangleleft \mathbf{B}$ , if `A` preserves the observable of `B`. The notion of semantic subclassing is useful for exploring the expressive power of specification languages and the modular verification of object-oriented programs.

### 4.1 Syntactic Subclassing

The intuition behind the syntactic subclassing relation is inspired by the Smalltalk approach to inheritance: a subclass must 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 3 (Syntactic Subclassing).** *Let `A` and `B` be two classes,  $\iota(\cdot)$  as in Def. 2. Then the syntactic subclass relation is defined as  $\mathbf{A} \blacktriangleleft \mathbf{B} \iff \iota(\mathbf{A}) \supseteq \iota(\mathbf{B})$ .*

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

*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 by using the syntactic construct “`A extends B { extension }`”, where `A` is the name of the subclass and `extension` are the fields 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  $\mathbf{A} \blacktriangleleft \mathbf{B}$  always holds.  $\square$

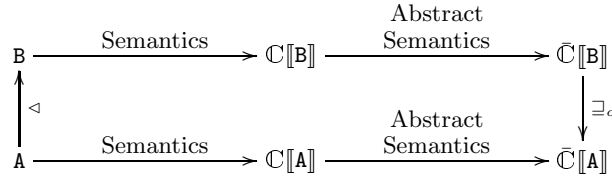
## 4.2 Semantic Subclassing

The semantic subclassing relation formalizes the intuition that up-to a given property, a class  $A$  behaves like a class  $B$ . 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, semantic subclassing can be defined in terms of the preservation of observables. In fact, as  $\sqsubseteq_o$  is the abstract counterpart for 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 4 (Semantic Subclassing).** *Let  $\langle O, \sqsubseteq_o \rangle$  be an abstract domain of observables and let  $A$  and  $B$  be two classes. Then the semantic subclassing relation with respect to  $O$  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`.  $\square$

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 [2].



**Fig. 5.** A visualization of the semantic subclassing relation

The relation between classes, concrete semantics and observables can be visualized by the diagram in Fig. 5. When the abstract semantics of  $A$  and  $B$  are compared, that of  $A$  implies the one of  $B$ . This means that  $A$  refines  $B$  w.r.t. the properties encoded by the abstract domain  $O$ , in accord with the mundane approach of inheritance where a subclass is as a specialization of its ancestors [25].

The next lemma states the monotonicity of  $\triangleleft$  w.r.t. the observed properties:

**Lemma 2.** *Let  $A$  and  $B$  be classes,  $\langle P, \sqsubseteq \rangle$  and  $\langle P', \sqsubseteq' \rangle$  be abstract domains in  $\mathcal{A}(\mathcal{P}(\Sigma))$  such that  $\langle P, \sqsubseteq \rangle \leq \langle P', \sqsubseteq' \rangle$ . If  $A \triangleleft_{O[P]} B$  then  $A \triangleleft_{O[P']} B$ .*

By Lemma 2, the more precise the domain of observables, the more precise the induced subclass relation. If we *observe* a more precise property about the class semantics then we are able to better *distinguish* between the different classes.

*Example 4.* Let us consider the 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 Lemma 2. Observe that in general the converse is not true: for instance  $\mathcal{H}_2$  is not admissible for congruences.  $\square$

When considering the *identity* Galois connection  $\langle \lambda x. x, \lambda x. x \rangle$ , Def. 4 above boils down to the observation of the concrete semantics, so that by Lemma 2,  $\triangleleft_{O[\mathcal{P}(\Sigma)]}$  is the most precise semantic subclassing relation. Furthermore, the semantic subclass relation induced by the most abstract domain is the trivial one, in which all classes are in relation with all others. As a consequence, given two classes A and B there always exist an abstract domain of observables  $O$  such that  $A \triangleleft_O B$ . However, in general there not exists a *least* domain of observables such that the two are in the semantic subclass relation, 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 as:

<pre> A.m() {   x = 1; y = 2;   if (x &gt; 0) &amp;&amp; (y % 2 == 0) {     x = 1; y = 4; }   else {     x = 1; y = 8; }}                 </pre>	<pre> B.m() {   x = 1; y = 2;   if (x &gt; 0) &amp;&amp; (y % 2 == 0) {     x = 1; y = 2; }   else {     x = 3; y = 10; }}                 </pre>
--	---

When considering the domain of intervals [10] as observables, we infer that  $A \triangleleft_{\text{Intervals}} B$  as  $([1, 1], [4, 8]) \sqsubseteq ([1, 3], [2, 10])$  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, [11], Intervals  $\times$  Parities we have that  $A \not\triangleleft_{\text{Intervals} \times \text{Parities}} B$  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 Lemma 2 it follows that it cannot be smaller, too.  $\square$

*Observation.* The previous example emphasizes a strict link between the concept of subtyping in specification languages for object-oriented programs and the notion of abstraction. Let us consider two classes A and B, two specification languages  $\mathcal{L}_1$  and  $\mathcal{L}_2$ , and the *strongest* properties we can express about the behavior of A and B in  $\mathcal{L}_1$  and in  $\mathcal{L}_2$ , say respectively  $\varphi_1^A, \varphi_1^B$  and  $\varphi_2^A, \varphi_2^B$ . Let us suppose that  $\varphi_1^A \Rightarrow \varphi_1^B$  and  $\varphi_2^A \Rightarrow \varphi_2^B$ . By definition of behavioral subtyping, [18], A is a subclass of B in both  $\mathcal{L}_1$  and in  $\mathcal{L}_2$ . Nevertheless, by Ex. 5, by the definition of observable of a class, and by basic abstract interpretation theory [8], it follows that if we consider a specification language  $\mathcal{L}_3$  expressive enough to contain both  $\mathcal{L}_1$  and  $\mathcal{L}_2$ , and the corresponding *strongest* properties  $\varphi_3^A, \varphi_3^B$ , then  $\varphi_3^A \not\Rightarrow \varphi_3^B$ . This means that when a more expressive language is used then the classes A and B are no more related. This fact enlightens an interesting trade-off between the expressive power of specification languages for object-oriented programs and the granularity of the subtyping relation they support.

### 4.3 Modular Verification of Polymorphic Code

Thanks to the following lemma, the notion of semantic subclass turns out to be useful for the modular verification of polymorphic code:

**Lemma 3 (Modular Verification).** *Let  $\langle P, \sqsubseteq \rangle$  be an abstract domain, let  $g \in [O[P] \rightarrow P]$  be a monotonic function and let  $f \in [\mathcal{C} \rightarrow P]$  be defined as  $f = \lambda \mathbf{B}. g(\bar{\mathcal{C}}[\mathbf{B}])$ . Then,  $\mathbf{A} \triangleleft_{O[P]} \mathbf{B}$  implies that  $f(\mathbf{A}) \sqsubseteq f(\mathbf{B})$ .*

Let us consider a function “ $\mathbf{m}(\mathbf{B} \ \mathbf{b}) \ \{ \ \mathbf{body}_{\mathbf{m}} \}$ ”. The best abstract semantics, [11], of  $\mathbf{m}$  w.r.t. a given abstract domain  $\langle P, \sqsubseteq \rangle$  is  $\bar{\mathbb{s}}[\mathbf{m}]$ <sup>2</sup>. By Galois connection properties,  $\bar{\mathbb{s}}[\mathbf{m}]$  is a monotonic function. Let  $o \in O[P]$ . We define  $g$  as the function obtained from  $\bar{\mathbb{s}}[\mathbf{m}]$  by replacing each occurrence of an invocation of a method of  $\mathbf{b}$ , e.g.  $\mathbf{b.n}(\dots)$ , inside  $\mathbf{body}_{\mathbf{m}}$  with the corresponding preconditions and post-conditions of  $o$  [14]. We denote it with  $\mathbf{m}[\mathbf{b} \mapsto o]$ . Hence,  $g = \lambda o. \bar{\mathbb{s}}[\mathbf{m}[\mathbf{b} \mapsto o]]$  is a monotonic function, and in particular  $\bar{\mathbb{s}}[\mathbf{m}] \sqsubseteq g(\bar{\mathcal{C}}[\mathbf{B}])$  as  $\bar{\mathcal{C}}[\mathbf{B}]$  is an approximation of the behavior of  $\mathbf{b}$  in all the possible contexts. Then, we can apply Lemma 3 so that for every class  $\mathbf{A}$ ,  $\mathbf{A} \triangleleft_{O[P]} \mathbf{B}$ , we have that  $g(\bar{\mathcal{C}}[\mathbf{A}]) \sqsubseteq g(\bar{\mathcal{C}}[\mathbf{B}])$ . As a consequence, if we can prove that  $g(\bar{\mathcal{C}}[\mathbf{B}]) \sqsubseteq \bar{\mathbf{S}}$  for a given specification  $\bar{\mathbf{S}}$ , by transitivity, it follows that  $g(\bar{\mathcal{C}}[\mathbf{A}]) \sqsubseteq \bar{\mathbf{S}}$ , for every semantic subclass  $\mathbf{A} \triangleleft_{O[P]} \mathbf{B}$ , i.e.,  $\mathbf{m}$  is correct w.r.t. the specification  $\bar{\mathbf{S}}$ .

*Example 6.* Consider the function `inv` in Sect. 2. We want to prove the property that `inv` never performs a division by zero. Let us instantiate  $P$  with the parity abstract domain. By Ex. 1 we know that  $\mathbf{x} = e$ . By an abstract evaluation of the `return` expression, one obtains  $1/(1 - e\%2) = 1/d$ , that is always defined (as obviously zero is not an odd number). As a consequence, when an instance of `Even` is passed to `inv`, it does not throw any division-by-zero exception. Furthermore, for what said above, this is true for all the semantic subclasses of `Even`.  $\square$

### 4.4 Relation Between $\blacktriangleleft$ and $\triangleleft$

Consider two classes  $\mathbf{A}$  and  $\mathbf{B}$  such that  $\mathbf{A} \blacktriangleleft \mathbf{B}$ . By definition, this means that all the names (fields or methods) defined in  $\mathbf{B}$  are defined in  $\mathbf{A}$  too. In general, such a condition is too weak to state something “*interesting*” about the semantics of  $\mathbf{A}$  w.r.t. that of  $\mathbf{B}$ : as seen before, there exists a domain of observables  $O$  such that  $\mathbf{A} \triangleleft_O \mathbf{B}$ , and in most cases such a domain is the most abstract one, and by Lemma 2 this implies that  $\triangleleft$  is a uninteresting 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. If the constructor of a class  $\mathbf{A}$  is *compatible* with that of  $\mathbf{B}$ , and if the methods of  $\mathbf{A}$  do not violate the class invariant of  $\mathbf{B}$ , then  $\mathbf{A}$  is a semantic subclass of  $\mathbf{B}$ . On the other hand, semantic subclassing *almost* implies syntactic subclassing. This is formalized by the following theorems [24]:

<sup>2</sup> We consider the best abstract function in order to simplify the exposition. Nevertheless the paper’s results still hold when a generic upper-approximation is considered.

**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  $A \blacktriangleleft B$ , and let  $(P, \sqsubseteq) \in \mathcal{A}(\mathcal{P}(\Sigma))$ . If (i)  $I_B$  is a class invariant for  $B$ , (ii)  $\mathbb{M}^{\triangleright}[\![\text{init}_A]\!] \sqsubseteq \mathbb{M}^{\triangleright}[\![\text{init}_B]\!]$ , (iii)  $\forall \bar{S} \in P. \forall m \in M_A \cap M_B. \mathbb{M}^{\triangleright}[\![m]\!](\bar{S}) \sqsubseteq I_B$  and (iv)  $\forall m \in M_A. m \notin M_B \implies \mathbb{M}^{\triangleright}[\![m]\!](\bar{S}) \sqsubseteq I_B$  then  $A \triangleleft_{O[P]} B$ .

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

## 5 Meaning-Preserving Manipulation of Class Hierarchies

In this Section, we exploit the results of the previous sections to introduce the concept of admissible class hierarchy, and to define and prove correct some operators on class hierarchies.

### 5.1 Admissible Semantic Class Hierarchies

For basic definitions on trees, the reader may refer to [6]. If  $T$  is a tree,  $\text{nodesOf}(T)$  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  $\text{tree}(r, S)$ .

Here we only consider single inheritance so that class hierarchies are trees of classes. An admissible hierarchy w.r.t. a transitive relation  $\rho$  on classes is a tree such that all the nodes 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 5 (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 a class hierarchy which is admissible w.r.t.  $\rho$ , if (i)  $\text{nodesOf}(\mathcal{H}) \subseteq \mathcal{C}$ , and (ii)  $\forall n \in \text{nodesOf}(\mathcal{H}). \forall n' \in \text{sonsOf}(n). n' \rho n$ .

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 of mainstream object-oriented languages. In fact, when  $\rho$  is instantiated with  $\blacktriangleleft$ , we obtain class hierarchies in which all the subclasses have at least the same methods as their superclass. A *semantic class hierarchy* is just the instantiation of the Def. 5 with the relation  $\triangleleft$ . The theorems and lemmata of the previous sections can be easily lifted to class hierarchies:

*Example 7.* Consider the two hierarchies in Fig. 2.  $\mathcal{H}_1$  is admissible w.r.t.  $\triangleleft_{\text{Con}}$  and  $\mathcal{H}_2$  is admissible w.r.t.  $\triangleleft_{\text{Parities}}$  but not w.r.t.  $\triangleleft_{\text{Con}}$ .  $\square$

In order to manipulate hierarchies we wish to preserve admissibility. This is why we need the notion of a *fair* operator. A fair operator on class hierarchies transforms a set of class hierarchies admissible w.r.t. a relation  $\rho$  into a class hierarchy that is admissible w.r.t. a relation  $\rho'$ .

**Definition 6 (Fair Operator).** Let  $\rho$  and  $\rho'$  be transitive relations. Then we say that a function  $\tau$  is a fair operator w.r.t.  $\rho$  and  $\rho'$  if  $\tau \in [\mathcal{P}(\mathbb{H}[\rho]) \rightarrow \mathbb{H}[\rho']]$ .

In the following, when not stated otherwise, we assume that  $\rho = \rho' = \triangleleft$ .

## 5.2 Class Insertion

The first fair 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. 6. It uses as sub-routine CSS, an algorithm for computing a common semantic superclass of two given classes, that is depicted in Fig. 7 and discussed in the next section.

The insertion algorithm takes as input an admissible class hierarchy  $\mathcal{H}$  and a class  $C$ . Four cases are distinguished. (i) if  $C$  already belongs to  $\mathcal{H}$  then the hierarchy keeps unchanged. (ii) If  $C$  is a superclass of the root of  $\mathcal{H}$ , then a new class hierarchy whose root is  $C$  is returned. (iii) If  $C$  is a subclass of the root of  $\mathcal{H}$ , then the insertion must preserve the *admissibility* of the hierarchy. If  $C$  is a superclass of some of the successors, then it is inserted between the root of  $\mathcal{H}$  and such successors. Otherwise it checks 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. (iv) If  $C$  and the root of  $\mathcal{H}$

```

 $\mathcal{H} \uplus C \triangleq$  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  $\text{tree}(C, R)$ 
if  $C \triangleleft R$  then
  if  $\mathcal{H}_< \neq \emptyset$  then
    return  $\text{tree}(R, (S - \mathcal{H}_<) \cup \text{tree}(C, \mathcal{H}_<))$ 
  if  $\mathcal{H}_> \neq \emptyset$  then select  $\mathcal{K} \in S$ 
    return  $\text{tree}(R, (S - \mathcal{K}) \cup (\mathcal{K} \uplus C))$ 
  else return  $\text{tree}(R, S \cup \{C\})$ 
else select  $C_{\top} = \text{CSS}(R, C)$ 
return  $\text{tree}(C_{\top}, \{R, C\})$ 

```

**Fig. 6.** The algorithm for a fair class insertion

```

 $\text{CSS}(A, B) \triangleq$  let  $A = \langle F_A, \text{init}_A, M_A \rangle$ ,
   $B = \langle F_B, \text{init}_B, M_B \rangle$ ,
   $F_C = \emptyset$ ,  $\text{init}_C = \text{init}_A$ ,  $M_C = \emptyset$ 
repeat
  select  $f \in F_A - F_C$ 
  if  $B \triangleleft \langle F_C \cup \{f\}, \tau_{F_C \cup \{f\}}(\text{init}_A), \tau_{F_C \cup \{f\}}(M_C) \rangle$ 
    then  $F_C = F_C \cup \{f\}$ ,
       $\text{init}_C = \tau_{F_C \cup \{f\}}(\text{init}_A)$ 
  || select  $m \in M_A - M_C$ 
  if  $B \triangleleft \langle F_C, \text{init}_C, \tau_{F_C}(M_C \cup \{m\}) \rangle$ 
    then  $M_C = M_C \cup \{m\}$ 
until no more fields or methods are added
return  $\langle F_C, \text{init}_C, \tau_{F_C}(M_C) \rangle$ 

```

**Fig. 7.** Algorithm for computing the CSS

are unrelated, the algorithm returns a new hierarchy whose root is a superclass of both  $\mathcal{C}$  and the root of  $\mathcal{H}$ .

The soundness of the algorithm follows from the observation that, if in the input hierarchy there is an admissible path from a class  $B$  to a class  $A$ , then in the extended hierarchy there still exists an admissible path from  $B$  to  $A$ .

**Lemma 4 (Soundness of  $\uplus$ , [24]).** *The operator  $\uplus$  defined in Fig. 6 is a fair operator w.r.t.  $\triangleleft$ , i.e.,  $\uplus \in [\mathbb{H}[\triangleleft] \times \mathcal{C} \rightarrow \mathbb{H}[\triangleleft]]$ .*

*Example 8.* Consider the hierarchy  $\mathcal{H}_1$  and the classes `MultiFour` and `MultiTwenty` of Sect.2.  $(\mathcal{H}_1 \uplus \text{MultiFour}) \uplus \text{MultiTwenty} = \mathcal{H}_3$  of Fig.4.  $\square$

Because of Th. 1, the algorithm  $\uplus$  is effective as soon as the underlying domain of observables is suitable for a static analysis, i.e. the abstract elements are computer representable, the order on  $P$  is decidable, and a widening operator ensures the convergence of the fixpoint computation. The dual operator, i.e., the elimination of a class from a hierarchy, corresponds straightforwardly to the algorithm for removing a node from an ordered tree [6].

### 5.3 Common Semantic Superclass

From the previous section we were left to define (and prove correct) the algorithm that returns the common *semantic* superclass (CSS) of two given classes. First we recall the definition of meaning-preserving transformation  $\tau$  [12]:

**Definition 7 (Program Transformation).** *Let  $A = \langle F, \text{init}, M \rangle$  and  $\langle \alpha, \gamma \rangle$  a Galois connection satisfying (2). A meaning-preserving program transformation  $\tau \in [F \rightarrow M \rightarrow M]$  is such that  $\forall f \in F. \forall m \in M$ : (i)  $\tau_f(m)$  does not contain the field  $f$  and (ii)  $\forall d \in P. \alpha(\mathbb{M}^{\triangleright}[\![m]\!])(\gamma(d)) \sqsubseteq \alpha(\mathbb{M}^{\triangleright}[\![\tau_f(m)]\!])(\gamma(d))$ .*

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. 7. It is parameterized by the underlying abstract domain of observables and a meaning preserving map  $\tau$ . The algorithm 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 5 (Soundness of CSS).** *Let  $A$  and  $B$  be two classes. Then  $\text{CSS}(A,B)$  is such that  $A \triangleleft \text{CSS}(A,B)$  and  $B \triangleleft \text{CSS}(A,B)$ .*

It is worth noting that in general,  $\text{CSS}(A,B) \neq \text{CSS}(B,A)$ . 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 [20].

*Example 9.* Consider the classes `MultiEight` and `MultiTwelve` and `MultiFour` defined as in Sect. 2. When using the abstract domain of linear congruences,  $\text{CSS}(\text{MultiEight}, \text{MultiTwelve}) = \text{MultiFour}$ .  $\square$

## 5.4 Merging of Hierarchies

The last refactoring operation on hierarchies we consider is about merging. The algorithm  $\uplus$  can be used as a basis for the algorithm to merge two admissible class hierarchies:

```

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

```

**Lemma 5.**  $\uplus$  is a fair operator w.r.t.  $\triangleleft$ , i.e.,  $\uplus \in [\mathbb{H}[\triangleleft] \rightarrow \mathbb{H}[\triangleleft]]$ .

It is worth mentioning that the modularity and modulability of the operators described in this section are the crucial keys that allow to apply them also to “real world” hierarchy management issues [33].

## 6 Related Work

In their seminal work on Simula [13], Dahl and Nygaard justified the concept of inheritance on syntactic bases, namely as textual concatenation of program blocks. A first semantic approach is [15] an (informal) operational approach to the semantics of inheritance is introduced. In particular the problem of specifying the semantics of message dispatch is reduced to that of method lookup. In [5] a denotational characterization of inheritance is introduced and proved correct w.r.t. an operational semantics based on the method lookup algorithm of [15]. An unifying view of the different forms of inheritance provided by programming languages is presented in [1]. In the *objects as records model* [2], the semantics of an object is abstracted with its type: inheritance is identified with subtyping. Such an approach is not fully satisfactory as shown in [4]. The notion of subtyping has been generalized in [18] where inheritance is seen 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. The main difference 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.

As for class hierarchies refactoring, [32] presents a semantics-preserving approach to class composition. Such an approach preserves the behavior of the composing hierarchies when they do not interfere. If they do interfere, a static analysis determines which components (classes, methods, etc.) of the hierarchies may interfere, given a set of programs that use such hierarchies. Such an approach is the base of the [33], 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 a 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 the domain of observables to be the concrete one.

## 7 Conclusions and Future Work

We introduced a framework for the definition and the manipulation of class hierarchies based on semantics abstraction. The main novelty of this approach is twofold: it provides a logic-based solid foundation of class refactoring operations that are safe by construction, and allows us to tune it according to the observed property. The next goal is the development of a tool for the semi-automatic refactoring of class hierarchies, based on [19,21], and the design of abstract domains capturing properties expressible in JML [17].

## Acknowledgments

Thanks to Radhia Cousot for her kind support. This work was partly supported by École Polytechnique, and by MIUR projects COFIN 2004013015 - FIRB RBAU018RCZ.

## References

1. 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.
2. 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.
3. G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.
4. 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.
5. W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, November 1994.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stei. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
7. A. Cortesi and F. Logozzo. Abstract interpretation-based verification of non functional requirements. In *Proceedings of the 7th International Conference on Coordination Models and Languages (COORD'05)*, volume 3654 of *Lecture Notes in Computer Science*, pages 49–62. Springer-Verlag, April 2005.
8. P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier Science, 1990.

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. 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.
13. O. Dahl and K. Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM (CACM)*, 9(9):671–678, September 1966.
14. D. L. Detlefs, K. Rustan M. Leino, G. Nelson, and Saxe J.B. Extended static checking. Research Report #159, Compaq Systems Research Center, Palo Alto, USA, December 1998.
15. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
16. 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.
17. G. T. Leavens, A. L. Baker, and C. Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, November 2003.
18. 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.
19. F. Logozzo. Class-level modular analysis for object oriented languages. In *Proceedings of the 10th Static Analysis Symposium 2003 (SAS '03)*, volume 2694 of *Lectures Notes in Computer Science*, pages 37–54. Springer-Verlag, June 2003.
20. 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.
21. 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.
22. F. Logozzo. *Modular Static Analysis of Object-oriented languages*. PhD thesis, École Polytechnique, 2004.
23. F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems and Structures*, 2005 (to appear).
24. F. Logozzo and A. Cortesi. Semantic class hierarchies by abstract interpretation. Technical Report CS-2004-7, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy, 2004.
25. B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.
26. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

27. NetBeans.org and Sun Microsystems, Inc. Netbeans IDE, 2004.
28. P. Odifreddi. *Classical Recursion Theory*. Elsevier, Amsterdam, 1999.
29. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.
30. 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.
31. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–288, 2002.
32. 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.
33. M. Streckenbach 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.