Separate Compositional Analysis of Class-based Object-oriented Languages

Francesco Logozzo

STIX - Ecole Polytechnique
F-91128 Palaiseau (France)
Francesco.Logozzo@polytechnique.fr

Abstract. We present a separate compositional analysis for object-oriented languages. We show how a generic static analysis of a context that uses an object can be split into two separate semantic functions involving respectively only the context and the object. The fundamental idea is to use a regular expressions for approximating the interactions between the context and the object. Then, we introduce an iterative schema for composing the two semantic functions. A first advantage is that the analysis can be parallelized, with a consequent gain in memory and time. Furthermore, the iteration process returns at each step an upper approximation of the concrete semantics, so that the iterations can be stopped as soon as the desired degree of precision is reached. Finally, we instantiate our approach to a core object-oriented language with aliasing.

1 Introduction

One important facet of object-oriented design is encapsulation. Encapsulation hides the objects' inner details from the outside world and allows a hierarchical structuring of code. As a consequence, a program written in the object-oriented style has often the structure of $\mathtt{C}[\mathtt{o}]$, where $\mathtt{C}[\cdot]$ is a context which interacts with an encapsulated object $\mathtt{o}$. The interaction between the context and the object can be of two kinds. The first, direct, one is through method invocations. The context invokes a method of the object which may return some value and modify its internal state. In particular, the value returned by the method can be a pointer to the value of a field. Thus, the object may expose a part of its internal state to the context, which can arbitrarily change the value of the field. We call this second kind of interaction an indirect one.

We are interested in an analysis that exploits the encapsulation features of the object-oriented languages, so that the context and the object can be analyzed separately. In fact, most available analyses are not separated e.g. [7], or they are imprecise as they assume the worst case for the calling context, e.g. [8]. A separate analysis presents several advantages. First, it may significantly reduce the overall analysis cost both in time and space, as e.g. different computers can be used for the analysis of the context and the object. Second, as the total memory consumption is reduced, very precise analyses can be used for the context and/or

the object. Third, it allows a form of modular analysis: if o is replaced by another object o' then the analysis of $\mathsf{C}[\mathsf{o}']$ requires just the analysis of o'. For instance, this is the case when $\mathsf{C}[\cdot]$ is a function and o and o' are actual parameters, or when o' is a refinement of o, e.g. o' is a sub-object of o.

We present a compositional separate analysis of class-based object oriented languages. We illustrate and prove our results for a core object-oriented language with aliasing. In particular in the considered language the identity of an object is given by the memory address where its environment is stored. This implies that we handle objects aliasing and that objects are semantic rather than syntactic entities. This is in line with mainstream object-oriented languages, so the presented framework can be easily extended to cope with realistic languages.

In Sect. 2, we define the syntax and the concrete semantics for our language and in Sect. 3 we present a generic *monolithic* static analysis of the context and the object $[\![\mathsf{C}[\mathsf{o}]]\!]^{\mathsf{a}}$, parameterized by an abstract domain D^{a}. In Sect. 4, we show how it can be split into two semantic functions, Γ and Θ, corresponding respectively to the analysis of the context and the object. The fundamental idea is the use of regular expressions for approximating the interactions between the context and the object. Therefore, we refine the abstract domain D^{a} with a domain of regular expressions. We have that:

- The object analysis Θ is a function that takes as input a map from objects to regular expressions. It returns a map from objects to their approximations.
- The context analysis Γ is a function that takes as input the approximation of the semantics of the objects. It returns an abstract value and a map from objects to regular expressions.

The functions Θ and Γ are mutually recursive. Thus, we handle this situation with the usual iterative approach. In particular, we begin by assuming the worst-case for the objects approximations and the contexts. Then, we show that the iterations form a chain of increasing precision, each step being a sound upper-approximation of $[\![\mathsf{C}[\mathsf{o}]]\!]^{\mathsf{a}}$. This implies that the iterations can be stopped as soon as the desired degree of precision is reached, enabling a trade-off between precision and cost.

2 Concrete Semantics

We begin by defining the syntax and the semantics of a minimal Java-like language. We make some simplifying assumptions. First, in order to simplify the notation we assume the existence of just one class. The generalization of the results to the case of an arbitrary number of classes is straightforward. Second, we distinguish between a context, for which we will give the detailed syntax, and a class, for which we give just the interface, i.e. the fields and the methods, but not the definition of the methods body. This is not restrictive, as the notion of context is relative. For example, a context that accesses an object o can be the body of a method of an object o'. Such an o' can be accessed by further context, so that the contexts can be encapsulated.

2.1 Syntax

The class syntax can be abstractly modeled as a triplet $\langle \texttt{init}, \texttt{F}, \texttt{M} \rangle$ where `init` is the class constructor, `F` is a set of variables and `M` is a set of function definitions. We do not require to have typed fields or methods and without any loss of generality we assume that a class has just a single constructor and each access to a field `f` is done through `set_f/get_f`.

The syntax of a context is quite standard, except that we distinguish three kinds of assignments: the assignment of the value of a side-effects free expression to a variable, the assignment of the address pointed by a variable to another one and the assignment of the return value of a method call to a variable. So, let `x`, $\texttt{x}_1$, $\texttt{x}_2$ and `o` be variables, let `E` be an expression and let `b` be a boolean expression. Then the language of contexts is generated by the following grammar:

$$\begin{aligned} \texttt{C} ::= \ & \texttt{A o} \ = \ \texttt{new A(E)} \mid \texttt{C}_1\texttt{;C}_2 \mid \texttt{skip} \mid \texttt{x} = \texttt{E} \mid \texttt{x}_1 = \texttt{x}_2 \\ & \mid \texttt{x} = \texttt{o.m(E)} \mid \texttt{if b then C}_1 \texttt{ else C}_2 \mid \texttt{while b do C}. \end{aligned}$$

`C` denotes an arbitrary context, $\texttt{C}[\cdot]$ denotes a context that may contain one or more objects and $\texttt{C}[\texttt{o}]$ denotes a context that uses an object `o`. However, as we allow aliasing of objects, we cannot give the formal definition of $\texttt{C}[\texttt{o}]$ on a strictly syntactic basis. Therefore, such a definition is postponed to the next section.

2.2 Semantic Domains

The first step for the specification of the concrete semantics is the definition of the concrete domain. In our case, we are interested in a domain that models the fact that an object has its own identity and environment. Moreover, we need to express object aliasing. In order to fulfill the above requirements, we consider a domain whose elements are pairs of environments and stores. An environment is a map from variables to memory addresses, $\mathsf{Env} = [\mathtt{Var} \rightarrow \mathsf{Addr}]$, and a store is a map from addresses to memory elements, $\mathsf{Store} = [\mathsf{Addr} \rightarrow \mathsf{Val}]$. A memory element can be a primitive value as well as an environment or an address, i.e. $\mathsf{Env} \subseteq \mathsf{Val}$ and $\mathsf{Addr} \subseteq \mathsf{Val}$. In such a setting, the identity of an object is the memory address where its environment is stored. Therefore, two distinct variables are aliases for an object if they reference the same memory address.

2.3 Object Semantics

We consider an input/output semantics for the class constructor and the methods. The semantics of the constructor is a function $\mathcal{I}[\![\texttt{init}]\!] \in [\mathsf{Val} \times \mathsf{Store} \rightarrow \mathsf{Env} \times \mathsf{Store}]$ which takes as input the constructor's actual parameter and a store. It returns the (initialized) object environment and the (possibly modified) store. It is worth noting that the constructor does not return any value to the context.

The semantics of a method `m` is a function $\mathcal{M}[\![\texttt{m}]\!] \in [\mathsf{Val} \times \mathsf{Env} \times \mathsf{Store} \rightarrow \mathsf{Val} \times \mathsf{Env} \times \mathsf{Store}]$. It takes as input the method's actual parameter, the object

$$
\begin{aligned}
\mathcal{C}[\![\texttt{A o = new A(E)}]\!] &= \lambda \mathsf{e}, \mathsf{s}.\mathit{let}\ v = \mathcal{E}[\![\texttt{E}]\!](\mathsf{e},\mathsf{s}), a = \mathsf{alloc}(\mathsf{s}),\\
&\qquad (\mathsf{e}_0,\mathsf{s}_0) = \mathcal{I}[\![\texttt{init}]\!](v,\mathsf{s}),\\
&\qquad \mathit{in}\ (\mathsf{e}[\mathsf{o} \mapsto a], \mathsf{s}_0[a \mapsto \mathsf{e}_0])\\
\mathcal{C}[\![\texttt{C}_1\texttt{;}\,\texttt{C}_2]\!] &= \lambda \mathsf{e}, \mathsf{s}.\mathcal{C}[\![\texttt{C}_2]\!](\mathcal{C}[\![\texttt{C}_1]\!](\mathsf{e},\mathsf{s})) \qquad \mathcal{C}[\![\texttt{skip}]\!] = \lambda \mathsf{e}, \mathsf{s}.(\mathsf{e},\mathsf{s})\\
\mathcal{C}[\![\texttt{x = E}]\!] &= \lambda \mathsf{e}, \mathsf{s}.(\mathsf{e}, \mathsf{s}[\mathsf{e}(\texttt{x}) \mapsto \mathcal{E}[\![\texttt{E}]\!](\mathsf{e},\mathsf{s})])\\
\mathcal{C}[\![\texttt{x}_1 = \texttt{x}_2]\!] &= \lambda \mathsf{e}, \mathsf{s}.(\mathsf{e}, \mathsf{s}[\mathsf{e}(\texttt{x}_1) \mapsto \mathsf{e}(\texttt{x}_2)])\\
\mathcal{C}[\![\texttt{x = o.m(E)}]\!] &= \lambda \mathsf{e}, \mathsf{s}.\mathit{let}\ v = \mathcal{E}[\![\texttt{E}]\!](\mathsf{e},\mathsf{s}), (v_0,\mathsf{e}_0,\mathsf{s}_0) = \mathcal{M}[\![\texttt{m}]\!](v,\mathsf{s}(\mathsf{e}(\mathsf{o})),\mathsf{s}),\\
&\qquad \mathit{in}\ (\mathsf{e}, \mathsf{s}_0[\mathsf{e}(\texttt{x}) \mapsto v_0, \mathsf{e}(\mathsf{o}) \mapsto \mathsf{e}_0])\\
\mathcal{C}[\![\texttt{if b then C}_1\ \texttt{else C}_2]\!] &= \lambda \mathsf{e}, \mathsf{s}.\mathit{if}\ \mathcal{B}[\![\texttt{b}]\!](\mathsf{e},\mathsf{s}) = \texttt{tt}\ \mathit{then}\ \mathcal{C}[\![\texttt{C}_1]\!](\mathsf{e},\mathsf{s})\ \mathit{else}\ \mathcal{C}[\![\texttt{C}_2]\!](\mathsf{e},\mathsf{s})\\
\mathcal{C}[\![\texttt{while b do C}]\!] &= \mathrm{lfp}\lambda\phi.\lambda \mathsf{e}, \mathsf{s}.\mathit{if}\ \mathcal{B}[\![\texttt{b}]\!](\mathsf{e},\mathsf{s}) = \texttt{tt}\ \mathit{then}\ \phi(\mathcal{C}[\![\texttt{C}]\!](\mathsf{e},\mathsf{s}))\ \mathit{else}\ (\mathsf{e},\mathsf{s})
\end{aligned}
$$

Fig. 1. Semantics of the context

environment and the store. It returns a (possibly void) value, the new object environment and the modified store. It is worth noting that as $\mathsf{Addr} \subseteq \mathsf{Val}$ the method may expose a part of the object's internal state to the context.

2.4 Context Semantics

We define the context semantics in denotational style, by induction on the syntax. The semantics of expressions and that of the boolean expressions are assumed to be side-effect free, such that $\mathcal{E}[\![\mathsf{e}]\!] \in [\mathsf{Env} \times \mathsf{Store} \to \mathsf{Val}]$ and $\mathcal{B}[\![\mathsf{b}]\!] \in [\mathsf{Env} \times \mathsf{Store} \to \{\mathsf{tt}, \mathsf{ff}\}]$. A function $\mathsf{alloc} \in [\mathsf{Store} \to \mathsf{Addr}]$ returns a fresh memory address. The semantics of a context, $\mathcal{C}[\![\mathtt{C}]\!] \in [\mathsf{Env} \times \mathsf{Store} \to \mathsf{Env} \times \mathsf{Store}]$ is given in Fig. 1.

Some comments on the context semantics. When a class `A` is instantiated, the initial value is evaluated, and the class constructor is invoked with that value and the store. The class constructor returns the environment e_0 of the new object and the modified store s_0. Then the environment and the store change, so that `o` points to the memory allocated for storing e_0. When a method of the object `o` is invoked, its environment is fetched from the memory and passed to the method. This implies that the method has no access to the caller environment, but only to that of the object it belongs to. In other words, the context has the burden of setting the right environment for a method call, so that the handling of `this` is somehow transparent to the callee. For the rest, the semantics in Fig. 1 is a quite standard denotational semantics. In particular, the loop semantics is handled by the least fixpoint operator on the flat Scott-domain $\mathsf{Env} \times \mathsf{Store} \cup \{\bot\}$.

Using the context semantics, we can formally define the writing $\mathtt{C}[\mathsf{o}]$, i.e. the context $\mathtt{C}[\cdot]$ that uses an object `o`. Let $(\mathsf{e}_0, \mathsf{s}_0) \in \mathsf{Env} \times \mathsf{Store}$, such that $\mathcal{C}[\![\mathtt{C}]\!](\mathsf{e}_0, \mathsf{s}_0) = (\mathsf{e}, \mathsf{s})$. Then a context `C` uses an object `o` if $\exists \mathtt{x} \in \mathtt{Var}.\mathsf{e}(\mathtt{x}) = \mathsf{o} \wedge \mathsf{s}(\mathsf{e}(\mathtt{x})) \in \mathsf{Env}$.

2.5 Collecting Semantics

A semantic property is a set of possible semantics of a program. The set of semantic properties $\mathcal{P}(\mathsf{Env} \times \mathsf{Store})$ is a complete boolean lattice $\langle \mathcal{P}(\mathsf{Env} \times \mathsf{Store}), \subseteq, \emptyset, \mathsf{Env} \times \mathsf{Store}, \cup, \cap \rangle$ for subset inclusion, that is logical implication. The standard collecting semantics of a program, $[\![\mathtt{C}]\!](In) = \{\mathcal{C}[\![\mathtt{C}]\!](\mathsf{e}, \mathsf{s}) \mid (\mathsf{e}, \mathsf{s}) \in In\}$, is the strongest program property. The goal of a static analysis is to find a computable approximation of $[\![\mathtt{C}]\!]$.

3 Monolithic Abstract Semantics

We proceed to the definition of a generic abstract semantics for the language presented in the previous section. First we consider the abstract semantic domains. Afterward, we present the abstract semantics for the class constructor and methods, and for the context.

3.1 Abstract Semantic Domains

The values in $\mathcal{P}(\mathsf{Val})$ are approximated by an abstract domain Val^a. The correspondence between the two domains is given by the Galois connection [2]:

$$\langle \mathcal{P}(\mathsf{Val}), \subseteq, \emptyset, \mathsf{Val}, \cup, \cap \rangle \underset{\alpha_v}{\overset{\gamma_v}{\leftrightarrows}} \langle \mathsf{Val}^\mathsf{a}, \sqsubseteq^\mathsf{a}_v, \bot^\mathsf{a}_v, \top^\mathsf{a}_v, \sqcup^\mathsf{a}_v, \sqcap^\mathsf{a}_v \rangle.$$

The set of abstract addresses is $\mathsf{Addr}^\mathsf{a} \subseteq \mathsf{Val}^\mathsf{a}$. We assume Addr^a to be a sublattice of Val^a. If $\mathsf{o} \in \mathsf{Addr}$ denotes an object in the concrete, then $\vartheta = \alpha_v(\{\mathsf{o}\})$ is the corresponding abstract address. On the other hand, ϑ stands for the set of concrete addresses $\gamma_v(\vartheta)$, which may contain several objects. Therefore, ϑ approximates all the objects in $\gamma_v(\vartheta)$. We call ϑ an abstract object.

The domain D^a abstracts the domain of concrete properties $\mathcal{P}(\mathsf{Env} \times \mathsf{Store})$ by means of a Galois connection:

$$\langle \mathcal{P}(\mathsf{Env} \times \mathsf{Store}), \subseteq, \emptyset, \mathsf{Env} \times \mathsf{Store}, \cup, \cap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{D}^\mathsf{a}, \sqsubseteq^\mathsf{a}, \bot^\mathsf{a}, \top^\mathsf{a}, \sqcup^\mathsf{a}, \sqcap^\mathsf{a} \rangle.$$

We call an element of D^a an abstract state. In general, the domain D^a is a relational abstraction of $\mathcal{P}(\mathsf{Env} \times \mathsf{Store})$. We consider two projections such that for each $\mathsf{d}^\mathsf{a} \in \mathsf{D}^\mathsf{a}$, $\mathsf{d}^\mathsf{a} \downarrow_\mathsf{e}$ and $\mathsf{d}^\mathsf{a} \downarrow_\mathsf{s}$ are, respectively, the projections of d^a on the environment and the store. We use the brackets $\lfloor \cdot \rfloor$ to denote the inverse operation of the projection, i.e. given an abstraction for the environment and the store it returns the abstract state. Moreover, some operations are defined on D^a: $\mathsf{alloc}^\mathsf{a}$, $\mathsf{assign}^\mathsf{a}$, true^a and $\mathsf{false}^\mathsf{a}$. The first one, $\mathsf{alloc}^\mathsf{a} \in [\mathsf{D}^\mathsf{a} \to \mathsf{Addr}^\mathsf{a}]$, is the abstract counterpart for memory allocation. It takes an approximation of the state and it returns an abstract address where the object environment can be stored. It satisfies the soundness requirement: $\forall \mathsf{d}^\mathsf{a} \in \mathsf{D}^\mathsf{a}.\{\mathsf{alloc}(\mathsf{s}) \mid (\mathsf{e}, \mathsf{s}) \in \gamma(\mathsf{d}^\mathsf{a})\} \subseteq \gamma_v(\mathsf{alloc}^\mathsf{a}(\mathsf{d}^\mathsf{a}))$.

The function $\mathsf{assign}^\mathsf{a} \in [\mathsf{D}^\mathsf{a} \times (\mathrm{Var} \times \mathsf{D}^\mathsf{a})^+ \to \mathsf{D}^\mathsf{a}]$ handles the assignment in the abstract domain. It takes as input an abstract state and a non-empty list of bindings from variables to values. It returns the new abstract state. With an abuse

of notation, we sometimes write $\mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{s}} \mapsto \mathsf{d}_0^{\mathsf{a}} \lfloor_{\mathsf{s}})$ to denote that the abstract store $\mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{s}}$ is updated by $\mathsf{d}_0^{\mathsf{a}} \lfloor_{\mathsf{s}}$. Moreover, $\mathsf{true}^{\mathsf{a}}$, $\mathsf{false}^{\mathsf{a}} \in [\mathsf{BExp} \times \mathsf{D}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$ are the functions that given a boolean expression and an abstract element d^{a} return an abstraction of the pairs $(\mathsf{e}, \mathsf{s}) \in \gamma(\mathsf{d}^{\mathsf{a}})$ that make the condition respectively true or false. For instance $\mathsf{true}^{\mathsf{a}}$ is such that:

$$\forall \mathsf{b} \in \mathsf{BExp}. \forall \mathsf{d}^{\mathsf{a}} \in \mathsf{D}^{\mathsf{a}}. \{(\mathsf{e}, \mathsf{s}) \mid \mathcal{B}[\![\mathsf{b}]\!](\mathsf{e}, \mathsf{s}) = \mathsf{tt}\} \cap \gamma(\mathsf{d}^{\mathsf{a}}) \subseteq \mathsf{true}^{\mathsf{a}}(\mathsf{b}, \mathsf{d}^{\mathsf{a}}).$$

3.2 Abstract Object Semantics

The abstract semantics for the class constructor and methods mimics the concrete one. Therefore, the abstract counterpart for the constructor semantics is a function $\mathcal{I}[\![\mathtt{init}]\!]^{\mathsf{a}} \in [\mathsf{Val}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$, which takes an abstract value and an abstract state and returns an abstract environment, that of the new object, and an abstract store. The abstract semantics for methods is a function $\mathcal{M}[\![\mathtt{m}]\!]^{\mathsf{a}} \in [\mathsf{Val}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}} \to \mathsf{Val}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}}]$. The input is an abstract value and an abstract state, and the output is an abstraction of the return value and a modified abstract state.

3.3 Monolithic Abstract Context Semantics

The abstract semantics for contexts is defined on the top of the abstract semantics for the expressions and the basic operations of the abstract domain D^{a}. In particular, the abstract semantics of expressions is $\mathcal{E}[\![\mathsf{e}]\!]^{\mathsf{a}} \in [\mathsf{D}^{\mathsf{a}} \to \mathsf{Val}^{\mathsf{a}}]$. It must satisfy the soundness requirement: $\forall (\mathsf{e}, \mathsf{s}) \in \mathsf{Env} \times \mathsf{Store}.\ \mathcal{E}[\![\mathsf{e}]\!](\mathsf{e}, \mathsf{s}) \in \gamma_v \circ \mathcal{E}[\![\mathsf{e}]\!]^{\mathsf{a}} \circ \alpha(\{(\mathsf{e}, \mathsf{s})\})$.

The generic abstract semantics mimics the concrete semantics. In particular, when a method $\mathtt{m}$ is invoked, the corresponding abstract function $\mathcal{M}[\![\mathtt{m}]\!]^{\mathsf{a}}$ is used. In practice, this means that the body of a method $\mathtt{m}$ is analyzed from scratch at each invocation. Therefore the encapsulation of the object w.r.t. context is not exploited in the analysis. We call such an abstract semantics a *monolithic* abstract semantics in order to differentiate it from the separate compositional abstract semantics that we will introduce in the next section.

Finally, the monolithic abstract context semantics $[\![\mathtt{C}]\!]^{\mathsf{a}} \in [\mathsf{D}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$ is defined in Fig. 2. The semantics in Fig. 2 is quite similar to the concrete one in Fig. 1. It is worth noting that the burden of handling the assignment is left to the underlying abstract domain D^{a}, and in particular to the function $\mathsf{assign}^{\mathsf{a}}$. We use the notation $\mathrm{lfp}^{\sqsubseteq}_{\mathsf{d}} \lambda x.F(x, z)$ to denote the least fixpoint w.r.t. the order $\sqsubseteq$, greater than d of the equation $F(x, z) = (x, y)$, for some z and y. Nevertheless, in general the abstract domains $\mathsf{Val}^{\mathsf{a}}$ and D^{a} may not respect the Ascending Chain Condition (ACC), so that the convergence of the analysis is enforced through the widening operators $\nabla_v^{\mathsf{a}} \in [\mathsf{Val}^{\mathsf{a}} \times \mathsf{Val}^{\mathsf{a}} \to \mathsf{Val}^{\mathsf{a}}]$ and $\nabla^{\mathsf{a}} \in [\mathsf{D}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$. The soundness of the above semantics is a consequence of the definitions of this section:

$$
\begin{aligned}
[\![\texttt{A o} \;=\; \texttt{new A(E)}]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.\mathit{let}\ \mathsf{v}^{\mathsf{a}} = \mathcal{E}[\![\texttt{E}]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}), \vartheta = \mathsf{alloc}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}),\\
&\qquad \mathsf{d_0}^{\mathsf{a}} = \mathcal{I}[\![\texttt{init}]\!]^{\mathsf{a}}(\mathsf{v}^{\mathsf{a}}, \mathsf{d}^{\mathsf{a}})\\
&\qquad \mathit{in}\ \mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \vartheta \mapsto \mathsf{d_0}^{\mathsf{a}} \lfloor_{\mathsf{e}}, \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{s}} \mapsto \mathsf{d_0}^{\mathsf{a}} \lfloor_{\mathsf{s}})\\
[\![\texttt{C}_1\texttt{;}\,\texttt{C}_2]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.[\![\texttt{C}_2]\!]^{\mathsf{a}}([\![\texttt{C}_1]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}})) \qquad [\![\texttt{skip}]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}.\mathsf{d}^{\mathsf{a}}\\
[\![\texttt{x} = \texttt{E}]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.\mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \texttt{x} \mapsto \mathcal{E}[\![\texttt{E}]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}))\\
[\![\texttt{x}_1 = \texttt{x}_2]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.\mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \texttt{x}_1 \mapsto \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{e}}(\texttt{x}_2))\\
[\![\texttt{x} = \texttt{o.m(E)}]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.\mathit{let}\ \mathsf{v}^{\mathsf{a}} = \mathcal{E}[\![\texttt{E}]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}), \vartheta = \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{e}}(\texttt{o}),\\
&\qquad (\mathsf{v_0}^{\mathsf{a}}, \mathsf{d_0}^{\mathsf{a}}) = \mathcal{M}[\![\texttt{m}]\!]^{\mathsf{a}}(\mathsf{v}^{\mathsf{a}}, \lfloor \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{s}}(\vartheta), \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{s}} \rfloor),\\
&\qquad \mathit{in}\ \mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \texttt{x} \mapsto \mathsf{v_0}^{\mathsf{a}}, \vartheta \mapsto \mathsf{d_0}^{\mathsf{a}} \lfloor_{\mathsf{e}}, \mathsf{d}^{\mathsf{a}} \lfloor_{\mathsf{s}} \mapsto \mathsf{d_0}^{\mathsf{a}} \lfloor_{\mathsf{s}})\\
[\![\texttt{if b then C}_1\ \texttt{else C}_2]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.[\![\texttt{C}_1]\!]^{\mathsf{a}}(\mathsf{true}^{\mathsf{a}}(\texttt{b}, \mathsf{d}^{\mathsf{a}})) \sqcup^{\mathsf{a}} [\![\texttt{C}_2]\!]^{\mathsf{a}}(\mathsf{false}^{\mathsf{a}}(\texttt{b}, \mathsf{d}^{\mathsf{a}}))\\
[\![\texttt{while b do C}]\!]^{\mathsf{a}} &= \lambda \mathsf{d}^{\mathsf{a}}.\mathsf{false}^{\mathsf{a}}(\texttt{b}, \mathrm{lfp}^{\sqsubseteq^{\mathsf{a}}}_{\mathsf{d}^{\mathsf{a}}} \lambda x.[\![\texttt{C}]\!]^{\mathsf{a}}(\mathsf{true}^{\mathsf{a}}(\texttt{b}, x)))
\end{aligned}
$$

Fig. 2. Monolithic abstract semantics

Theorem 1 (Soundness of $[\![\texttt{C}]\!]^{\mathsf{a}}$). *The monolithic context abstract semantics is a sound approximation of the concrete semantics:* $\forall In \in \mathcal{P}(\mathsf{Env} \times \mathsf{Store})$. $[\![\texttt{C}]\!](In) \subseteq \gamma \circ [\![\texttt{C}]\!]^{\mathsf{a}} \circ \alpha(In)$.

4 Separate Abstract Semantics

The abstract semantics $[\![\cdot]\!]^{\mathsf{a}}$ defined in the previous section does not take into account the encapsulation features of object-oriented languages, so that, for instance each time a method of an object is invoked, its body must be analyzed. In this section we show how to split $[\![\cdot]\!]^{\mathsf{a}}$ into two parts. The first part analyzes the context using an approximation of the object. The latter analyzes the object using an approximation of the context.

4.1 Regular Expressions Domain

The main idea for the separate analysis is to refine the abstract domain D^{a} with the abstract domain R of regular expressions over the infinite alphabet $(\{\texttt{init}\} \cup \mathcal{P}(\texttt{M})) \times \mathsf{Val}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}}$. Given an object, the intuition behind the refinement is to use a regular expression to abstract the method's invocations performed by the context. In particular, each *letter* in the alphabet represents a set of methods that can be invoked, an approximation of their input values and an approximation of the state. Such a regular expression is built during the analysis of the context. Then it is used for the analysis of the object.

The definition of the regular expressions in R is given by structural induction. The base cases are the *null* string ε and the letters l of the alphabet $(\{\texttt{init}\} \cup \mathcal{P}(\texttt{M})) \times \mathsf{Val}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}}$. Then, if r_1 and r_2 are regular expressions so are the concatenation $r_1 \cdot r_2$, the union $r_1 + r_2$ and the Kleene-closure r_1^*.

$$
\begin{array}{rlrl}
\top_r \nabla_r x = x \nabla_r \top_r = \top_r & & x \nabla_r \varepsilon = \varepsilon \nabla_r x = x \\
\langle \mathtt{m}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle \nabla_r \langle \mathtt{m}_1, \mathsf{v}^{\mathsf{a}}_1, \mathsf{s}^{\mathsf{a}}_1 \rangle = \langle \mathtt{m} \cup \mathtt{m}_1, \mathsf{v}^{\mathsf{a}} \nabla^{\mathsf{a}}_v \mathsf{v}^{\mathsf{a}}_1, \mathsf{s}^{\mathsf{a}} \nabla^{\mathsf{a}} \mathsf{s}^{\mathsf{a}}_1 \rangle & & (r_1 \cdot r_2) \nabla_r n = (r_1 \nabla_r n) \cdot r_2 \\
(r_1 + r_2) \nabla_r n = (r_1 \nabla_r n) + (r_2 \nabla_r n) & & r^* \nabla_r n = (r \nabla_r n)^* \\
(r_1 \cdot r_2) \nabla_r (r'_1 \cdot r'_2) = (r_1 \nabla_r r'_1) \cdot (r_2 \nabla_r r'_2) & & r^*_1 \nabla_r r^*_2 = (r_1 \nabla_r r_2)^* \\
(r_1 + r_2) \nabla_r (r'_1 + r'_2) = (r_1 \nabla_r r'_1) + (r_2 \nabla_r r'_2) & & \\
x \nabla_r y = \top_r \qquad \text{in all the other cases} & &
\end{array}
$$

Fig. 3. Widening on regular expressions

The language generated by a regular expression r is defined by structural induction:

$$
\begin{array}{rl}
\mathcal{L}(\langle \mathtt{ms}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle) = \{ \langle \mathtt{m}, v, s \rangle \mid \mathtt{m} \in \mathtt{ms}, v \in \gamma_v(\mathsf{v}^{\mathsf{a}}), s \in \gamma(\mathsf{s}^{\mathsf{a}}) \} & \mathcal{L}(\varepsilon) = \emptyset \\
\mathcal{L}(r_1 \cdot r_2) = \{ s_1 \cdot s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2) \} & \mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r^*) = \mathrm{lfp}^{\subseteq}_{\emptyset} \lambda X. \mathcal{L}(r) \cup \{ s_1 \cdot s_2 \mid s_1 \in X, s_2 \in \mathcal{L}(r) \}. &
\end{array}
$$

The order on regular expressions is a direct consequence of the above definition: $\forall r_1, r_2 \in \mathsf{R}. r_1 \sqsubseteq_r r_2 \iff \mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$. So, two expressions are equivalent if they generate the same language: $r_1 \equiv r_2 \iff \mathcal{L}(r_1) = \mathcal{L}(r_2)$. From now on, we consider all the operations and definitions on regular expressions modulo the equivalence $\equiv$. The expression $\top_r = \langle \{\mathtt{init}\} \cup \mathtt{M}, \top^{\mathsf{a}}_v, \top^{\mathsf{a}} \rangle^* \in \mathsf{R}$ stands for a context that may invoke any method, with any input value and with any memory configuration for a non-specified number of times. So, it gives no information. Thus, it is the largest element of $\langle \mathsf{R}, \sqsubseteq_r \rangle$. The join of two regular expressions is simply their union: $\forall r_1, r_2 \in \mathsf{R}. r_1 \sqcup_r r_2 = r_1 + r_2$. Similarly, the meet operator $\sqcap_r$ can be defined, so that $\langle \mathsf{R}, \sqsubseteq_r, \varepsilon, \top_r, \sqcup_r, \sqcap_r \rangle$ is a complete lattice.

The domain R does not satisfy the ACC, so we define the widening operator of Fig.3 to deal with strictly increasing chains of regular expressions. There are two intuitions behind the operator in Fig.3. The first one is to preserve the syntactic structure of the regular expressions between two successive iterations, so that the number of $\{\cdot, +, ^*\}$ does not increase. The second one is to propagate the ∇_r inside the regular expressions in order to use the widenings on $\mathsf{Val}^{\mathsf{a}}$ and D^{a}. Convergence is assured as $\mathtt{M}$ is a finite set, and ∇^{a}_v and ∇^{a} are widenings on the respective domains.

For the purpose of our analysis, we need to associate with each abstract address, i.e. a set of concrete objects, a regular expression that denotes the interaction of the context on it. As a consequence we consider the functional lifting[1] $\dot{\mathsf{R}} = [\mathsf{Addr}^{\mathsf{a}} \to \mathsf{R}]$. The order $\dot{\sqsubseteq}_r$ is defined pointwise: $\forall \dot{r}_1, \dot{r}_2 \in \dot{\mathsf{R}}. \dot{r}_1 \dot{\sqsubseteq}_r \dot{r}_2 \Leftrightarrow \forall \vartheta \in \mathsf{Addr}^{\mathsf{a}}. \dot{r}_1(\vartheta) \sqsubseteq_r \dot{r}_2(\vartheta)$. In a similar way, the join and the meet are defined

[1] We use the notation that given a domain D^{a}, $\dot{\mathsf{D}}^{\mathsf{a}}$ stands for the domain of functions $[\mathsf{Addr}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$. The operations on $\dot{\mathsf{D}}^{\mathsf{a}}$ are the pointwise extension of that of D^{a}: given an operation $\diamond$, then $\forall \dot{\mathsf{d}}^{\mathsf{a}}_1, \dot{\mathsf{d}}^{\mathsf{a}}_2 \in \dot{\mathsf{D}}^{\mathsf{a}}. \ \dot{\mathsf{d}}^{\mathsf{a}}_1 \mathbin{\dot{\diamond}} \dot{\mathsf{d}}^{\mathsf{a}}_2 = \lambda \vartheta. \dot{\mathsf{d}}^{\mathsf{a}}_1(\vartheta) \diamond \dot{\mathsf{d}}^{\mathsf{a}}_2(\vartheta)$.

$$
\begin{aligned}
\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(\varepsilon, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) =& \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle \\
\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(\langle \{\mathtt{init}\}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) =& \mathit{let}\ \langle \mathsf{e}_0^{\mathsf{a}}, \mathsf{s}_0^{\mathsf{a}} \rangle = \mathcal{I}[\![\mathtt{init}]\!]^{\mathsf{a}}(\mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \sqcup^{\mathsf{a}} \mathsf{i}^{\mathsf{a}}) \\
& \mathit{in}\ \langle \mathsf{i}^{\mathsf{a}} \sqcup^{\mathsf{a}} \mathsf{s}_0^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}}[\mathtt{init} \mapsto \langle \bot_v^{\mathsf{a}}, \mathsf{e}_0^{\mathsf{a}} \rangle] \rangle \\
\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(\langle \mathtt{ms}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) =& \mathit{let}\ \forall \mathtt{m_i} \in \mathtt{ms}.(\mathsf{v}_i^{\mathsf{a}}, \mathsf{s}_i^{\mathsf{a}}) = \mathcal{M}[\![\mathtt{m_i}]\!]^{\mathsf{a}}(\mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \sqcup^{\mathsf{a}} \mathsf{i}^{\mathsf{a}}),\ \langle \mathsf{w}_i^{\mathsf{a}}, \mathsf{q}_i^{\mathsf{a}} \rangle = \mathsf{p}^{\mathsf{a}}(\mathtt{m_i}) \\
& \mathit{in}\ (\mathsf{i}^{\mathsf{a}} \sqcup^{\mathsf{a}} \bigsqcup^{\mathsf{a}} \mathsf{s}_i^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}}[\mathtt{m_i} \mapsto \langle \mathsf{w}_i^{\mathsf{a}} \sqcup_v^{\mathsf{a}} \mathsf{v}_i^{\mathsf{a}}, \mathsf{q}_i^{\mathsf{a}} \sqcup^{\mathsf{a}} \mathsf{s}_i^{\mathsf{a}} \rangle]) \\
\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r_1 \cdot r_2, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) =& \mathit{let}\ (\mathsf{i}_1^{\mathsf{a}}, \mathsf{p}_1^{\mathsf{a}}) = \mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r_1, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle), (\mathsf{i}_2^{\mathsf{a}}, \mathsf{p}_2^{\mathsf{a}}) = \mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r_2, (\mathsf{i}_1^{\mathsf{a}}, \mathsf{p}_1^{\mathsf{a}})) \\
& \mathit{in}\ (\mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}}) \sqcup_o^{\mathsf{a}} (\mathsf{i}_1^{\mathsf{a}}, \mathsf{p}_1^{\mathsf{a}}) \sqcup_o^{\mathsf{a}} (\mathsf{i}_2^{\mathsf{a}}, \mathsf{p}_2^{\mathsf{a}}) \\
\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r_1 + r_2, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) =& \mathit{let}\ (\mathsf{i}_1^{\mathsf{a}}, \mathsf{p}_1^{\mathsf{a}}) = \mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r_1, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle), (\mathsf{i}_2^{\mathsf{a}}, \mathsf{p}_2^{\mathsf{a}}) = \mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r_2, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) \\
& \mathit{in}\ (\mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}}) \sqcup_o^{\mathsf{a}} (\mathsf{i}_1^{\mathsf{a}}, \mathsf{p}_1^{\mathsf{a}}) \sqcup_o^{\mathsf{a}} (\mathsf{i}_2^{\mathsf{a}}, \mathsf{p}_2^{\mathsf{a}}) \\
\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r^*, \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle) =& \mathrm{lfp}^{\sqsubseteq_o^{\mathsf{a}}}_{\langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle}\ \lambda x, y. \mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(r, (x, y))
\end{aligned}
$$

Fig. 4. Separate object abstract semantics

point-wise, so that $\langle \dot{\mathsf{R}}, \dot{\sqsubseteq}_r, \lambda\vartheta.\varepsilon, \lambda\vartheta.\top_r, \dot{\sqcup}_r, \dot{\sqcap}_r \rangle$ is a complete lattice. We call an element $\dot{r} \in \dot{\mathsf{R}}$ an interaction history.

4.2 Separate Object Analysis

The goal of the separate object analysis is to infer an object invariant and the method postconditions when the instantiation context is approximated by a regular expression. Thus, the input of the abstract semantics $\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}$ is a regular expression r and an initial abstract value for the object fields and the method preconditions. The output is an invariant for the object fields and the method postconditions, under the context represented by r. A postcondition is a pair consisting of an approximation of the return value and an abstract state. Thus, the result is an element of the abstract domain $\mathsf{O}^{\mathsf{a}} = \mathsf{D}^{\mathsf{a}} \times [\mathtt{M} \to \mathsf{Val}^{\mathsf{a}} \times \mathsf{D}^{\mathsf{a}}]$. From basic domain theory, the orders on D^{a} and $\mathsf{Val}^{\mathsf{a}}$ induce the order on O^{a}. So, the order is $\sqsubseteq_o^{\mathsf{a}} = \sqsubseteq^{\mathsf{a}} \times (\sqsubseteq_v^{\mathsf{a}} \dot{\times} \sqsubseteq^{\mathsf{a}})$, the least element is $\bot_o^{\mathsf{a}} = \langle \bot^{\mathsf{a}}, \lambda\mathtt{m}.\langle \bot_v^{\mathsf{a}}, \bot^{\mathsf{a}} \rangle \rangle$ and the largest $\top_o^{\mathsf{a}} = \langle \top^{\mathsf{a}}, \lambda\mathtt{m}.\langle \top_v^{\mathsf{a}}, \top^{\mathsf{a}} \rangle \rangle$. The meet, the join and the widening can be defined in a similar fashion, so that $\langle \mathsf{O}^{\mathsf{a}}, \sqsubseteq_o^{\mathsf{a}}, \bot_o^{\mathsf{a}}, \top_o^{\mathsf{a}}, \sqcup_o^{\mathsf{a}}, \sqcap_o^{\mathsf{a}} \rangle$ is a complete lattice. Finally, the separate object abstract semantics, $\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}} \in [\mathsf{R} \times \mathsf{O}^{\mathsf{a}} \to \mathsf{O}^{\mathsf{a}}]$, is defined in Fig. 4. Its definition is by structural induction on the regular expression r.

Some comments on the separate object semantics. The base cases are the empty expression ε and the letters $\langle \mathtt{ms}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle$ and $\langle \{\mathtt{init}\}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle$. In the first case the context does not perform any action, so that the state of the object does not change at all. In the latter, the context may invoke any method $\mathtt{m_i} \in \mathtt{ms}$. The abstract value $\bigsqcup^{\mathsf{a}} \mathsf{s_i}^{\mathsf{a}}$ approximates the object field values after calling the method $\mathtt{m}_1$ *or* $\mathtt{m}_2$ *or* ... *or* $\mathtt{m_n}$. As a consequence, $\mathsf{i}^{\mathsf{a}} \sqcup^{\mathsf{a}} \bigsqcup^{\mathsf{a}} \mathsf{s_i}^{\mathsf{a}}$ approximates the object fields before and after executing any method in $\mathtt{ms}$. Hence, it is an object invariant. On the other hand, if $\langle \mathsf{w}_i^{\mathsf{a}}, \mathsf{q}_i^{\mathsf{a}} \rangle$ is the initial approximation of the return values and the states reached after the execution of a method $\mathtt{m_i} \in \mathtt{ms}$, then $\langle \mathsf{w}_i^{\mathsf{a}} \sqcup_v^{\mathsf{a}} \mathsf{v}_i^{\mathsf{a}}, \mathsf{q}_i^{\mathsf{a}} \sqcup^{\mathsf{a}} \mathsf{s}_i^{\mathsf{a}} \rangle$

is the postcondition of $\mathtt{m_i}$ after its execution. The case of the constructor `init` is quite similar.

As for the inductive cases are concerned, the rules for concatenation and union formalize respectively that *"the context first performs r_1 and then r_2"* and *"the context can perform either r_1 or r_2"*. Finally, the rule for the Kleene-closure is a little bit more tricky. In fact the intuitive meaning of r^* is that, starting from an initial abstract value $\langle \mathsf{i}^\mathsf{a}, \mathsf{p}^\mathsf{a} \rangle$ the context performs the interaction encoded by r an unspecified number of times. We handle this case by considering the least fixpoint greater than $\langle \mathsf{i}^\mathsf{a}, \mathsf{p}^\mathsf{a} \rangle$ according to the order $\sqsubseteq^\mathsf{a}_o$ on O^a. If the abstract domains D^a and Val^a do not respect the ACC then the convergence of the iterations must be enforced using the following pointwise widening operator:

$$\lambda(\mathsf{i}^\mathsf{a}, \mathsf{p}^\mathsf{a}), (\mathsf{i}'^\mathsf{a}, \mathsf{p}'^\mathsf{a}).(\mathsf{i}^\mathsf{a} \nabla^\mathsf{a} \mathsf{i}'^\mathsf{a}, \lambda \mathtt{m}.\mathsf{p}^\mathsf{a}(\mathtt{m})\ (\nabla^\mathsf{a}_v \times \nabla^\mathsf{a})\ \mathsf{p}'^\mathsf{a}(\mathtt{m})).$$

The regular expression $r_\top = \langle \{\mathtt{init}\}, \top^\mathsf{a}_v, \top^\mathsf{a} \rangle \cdot \top_r$ stands for a context that calls at first the class constructor with an unknown value and then may invoke any object method, with any possible value, an unspecified number of times. Thus the abstract value $\langle \mathsf{i}^\mathsf{a}, \mathsf{p}^\mathsf{a} \rangle = \mathcal{O}[\![\vartheta]\!]^\mathsf{a}(r_\top, \bot^\mathsf{a}_o)$ is such that i^a is a property of the object fields valid for all the object instances, in any context. So it is a class invariant in the sense of [5,4]. In the following we refer to it as $[\![\mathtt{A}]\!]^\mathsf{a}$.

4.3 Separate Context Analysis

The separate context analysis $\mathcal{C}[\![\mathtt{C}[\cdot]]\!]^\mathsf{a}$ has two goals. The first goal is to analyze $\mathtt{C}[\mathtt{o}]$ without referring to the $\mathtt{o}$ code, but just to a pre-computed approximation of its semantics. The second goal is to infer, for each object $\mathtt{o}$ a regular expression r that describes the interaction of the context with $\mathtt{o}$. This r can then be used to refine the approximation of the object semantics. In general, a context creates several objects and it interacts with each of them in a different way. As a consequence, in the definition of the abstract context semantics $\mathcal{C}[\![\cdot]\!]^\mathsf{a}$ we use a domain $\dot{\mathsf{O}}^\mathsf{a} = [\mathsf{Addr}^\mathsf{a} \to \mathsf{O}^\mathsf{a}]$, whose elements are maps from abstract objects to their approximations. The definition of $\mathcal{C}[\![\mathtt{C}]\!]^\mathsf{a} \in [\mathsf{D}^\mathsf{a} \times \dot{\mathsf{O}}^\mathsf{a} \times \dot{\mathsf{R}} \to \mathsf{D}^\mathsf{a} \times \dot{\mathsf{R}}]$ is given by structural induction on $\mathtt{C}$ in Fig. 5.

Some comments on the separate context semantics. The semantics takes three parameters: an abstract state, an approximation of the semantics of the objects and the invocation history. When a class is instantiated, the semantics $\mathcal{C}[\![\cdot]\!]^\mathsf{a}$ (abstractly) evaluates the value to pass to the constructor `init` and it obtains an address ϑ for the new object. Then, it uses the object abstraction $\dot{\vartheta}(\vartheta)$ to get the constructor postcondition $\mathsf{p}^\mathsf{a}(\mathtt{init})$ and it updates the invocation history. In general, the abstract address ϑ identifies a set $\gamma_v(\vartheta)$ of concrete objects. So, the semantics adds an entry to the ϑ history corresponding to the invocation of `init`, with an input v^a and an abstract state d^a. Eventually, the result is the new abstract state, obtained considering the store after the execution of the constructor, and the updated invocation history. The sequence, the `skip` and the two assignments do not interact with objects so, in these cases, $\mathcal{C}[\![\cdot]\!]^\mathsf{a}$ is very close to the corresponding semantics of Fig. 2. The definition of

$$
\begin{aligned}
\mathcal{C}[\![\texttt{A o} \;=\; \texttt{new A(e)}]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ & \mathsf{v}^{\mathsf{a}} = \mathcal{E}[\![\texttt{e}]\!]^{\mathsf{a}}\mathsf{d}^{\mathsf{a}}, \vartheta = \mathsf{alloc}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}), \\
& \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle = \dot{\vartheta}(\vartheta), \langle \bot^{\mathsf{a}}_{v}, \mathsf{d}^{\mathsf{a}}_0 \rangle = \mathsf{p}^{\mathsf{a}}(\texttt{init}), \\
& \dot{r}' = \dot{r}[\vartheta \mapsto \langle \{\texttt{init}\}, \mathsf{v}^{\mathsf{a}}, \mathsf{d}^{\mathsf{a}} \rangle \sqcup_r \dot{r}(\vartheta)] \\
& \mathit{in}\ (\mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \vartheta \mapsto \mathsf{d}^{\mathsf{a}}_0 \downarrow_{\mathsf{e}}, \mathsf{d}^{\mathsf{a}} \downarrow_{\mathsf{s}} \mapsto \mathsf{d}^{\mathsf{a}}_0 \downarrow_{\mathsf{s}}), \dot{r}') \\
\mathcal{C}[\![\texttt{C}_1\texttt{;}\,\texttt{C}_2]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ & (\mathsf{d}^{\mathsf{a}}_1, \dot{r}_1) = \mathcal{C}[\![\texttt{C}_1]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}) \\
& \mathit{in}\ \mathcal{C}[\![\texttt{C}_2]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}_1, \dot{\vartheta}, \dot{r}_1) \\
\mathcal{C}[\![\texttt{skip}]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.(\mathsf{d}^{\mathsf{a}}, \dot{r}) \\
\mathcal{C}[\![\texttt{x} = \texttt{e}]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.(\mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \texttt{x} \mapsto \mathcal{E}[\![\texttt{e}]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}})), \dot{r}) \\
\mathcal{C}[\![\texttt{x}_1 = \texttt{x}_2]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.(\mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \texttt{x}_1 \mapsto \mathsf{d}^{\mathsf{a}} \downarrow_{\mathsf{e}}(\texttt{x}_2)), \dot{r}) \\
\mathcal{C}[\![\texttt{x} = \texttt{o.m(e)}]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ & \mathsf{v}^{\mathsf{a}} = \mathcal{E}[\![\texttt{e}]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}), \vartheta = \mathsf{d}^{\mathsf{a}} \downarrow_{\mathsf{e}}(\mathsf{o}), \\
& \langle \mathsf{i}^{\mathsf{a}}, \mathsf{p}^{\mathsf{a}} \rangle = \dot{\vartheta}(\vartheta), \langle \mathsf{v}^{\mathsf{a}}_{\mathtt{m}}, \mathsf{q}^{\mathsf{a}}_{\mathtt{m}} \rangle = \mathsf{p}^{\mathsf{a}}(\mathtt{m}) \\
& \mathsf{d}'^{\mathsf{a}} = \mathsf{assign}^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}, \texttt{x} \mapsto \mathsf{v}^{\mathsf{a}}_{\mathtt{m}}, \vartheta \mapsto \mathsf{q}^{\mathsf{a}}_{\mathtt{m}} \downarrow_{\mathsf{e}}, \mathsf{d}^{\mathsf{a}} \downarrow_{\mathsf{s}} \mapsto \mathsf{q}^{\mathsf{a}}_{\mathtt{m}} \downarrow_{\mathsf{s}}), \\
& \mathit{in}\ (\mathsf{d}'^{\mathsf{a}}, \dot{r}[\vartheta \mapsto \dot{r}(\vartheta) \cdot \langle \mathtt{m}, \mathsf{v}^{\mathsf{a}}, \lfloor \mathsf{d}^{\mathsf{a}} \downarrow_{\mathsf{s}}(\vartheta), \mathsf{d}^{\mathsf{a}} \downarrow_{\mathsf{s}} \rfloor \rangle]) \\
\mathcal{C}[\![\texttt{if b then C}_1 \texttt{ else C}_2]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ & (\mathsf{d}^{\mathsf{a}}_1, \dot{r}_1) = \mathcal{C}[\![\texttt{C}_1]\!]^{\mathsf{a}}(\mathsf{true}^{\mathsf{a}}(\mathtt{b}, \mathsf{d}^{\mathsf{a}}), \dot{\vartheta}, \dot{r}), \\
& (\mathsf{d}^{\mathsf{a}}_2, \dot{r}_2) = \mathcal{C}[\![\texttt{C}_2]\!]^{\mathsf{a}}(\mathsf{false}^{\mathsf{a}}(\mathtt{b}, \mathsf{d}^{\mathsf{a}}), \dot{\vartheta}, \dot{r}) \\
& \mathit{in}\ (\mathsf{d}^{\mathsf{a}}_1 \sqcup^{\mathsf{a}} \mathsf{d}^{\mathsf{a}}_2, \dot{r}_1 \dot{\sqcup}_r \dot{r}_2) \\
\mathcal{C}[\![\texttt{while b do C}]\!]^{\mathsf{a}} = \lambda \mathsf{d}^{\mathsf{a}}, \dot{\vartheta}, \dot{r}.\mathit{let}\ & (\mathsf{d}'^{\mathsf{a}}, \dot{r}') = \mathrm{lfp}^{\sqsubseteq^{\mathsf{a}} \times \dot{\sqsubseteq}_r}_{(\mathsf{d}^{\mathsf{a}}, \lambda \vartheta.\, \varepsilon)} \lambda(x, y).\mathcal{C}[\![\texttt{C}]\!]^{\mathsf{a}}(\mathsf{true}^{\mathsf{a}}(\mathtt{b}, x), \dot{\vartheta}, y) \\
& \mathit{in}\ (\mathsf{false}^{\mathsf{a}}(\mathtt{b}, \mathsf{d}'^{\mathsf{a}}), \lambda \vartheta.\dot{r}(\vartheta) \cdot (\dot{r}'(\vartheta))^*)
\end{aligned}
$$

Fig. 5. Separate context semantics

$\mathcal{C}[\![\cdot]\!]^{\mathsf{a}}$ for method invocation is similar to the constructor's one: it fetches the (abstract) address corresponding to `o` and the corresponding invariant. Then, it updates the abstract state, using the `m` postcondition, and the invocation history. The definition of the conditional merges the abstract states and the invocation histories originating from the two branches. Eventually, the loop is handled by the least fixpoint operator on the abstract domain $\mathsf{D}^{\mathsf{a}} \times \dot{\mathsf{R}}$. In particular we consider the least fixpoint greater than $(\mathsf{d}^{\mathsf{a}}, \lambda\vartheta.\, \varepsilon)$ as we need to compute an invocation history that is valid for all the iterations of the loop body. The history for the whole `while` command is the concatenation of the input history with the body one, repeated an unspecified number of times. As usual, the convergence of the analysis can be forced through the use of the widening operator $\lambda(\mathsf{d}^{\mathsf{a}}_1, \dot{r}_1).(\mathsf{d}^{\mathsf{a}}_2, \dot{r}_2),\ (\mathsf{d}^{\mathsf{a}}_1 \nabla^{\mathsf{a}} \mathsf{d}^{\mathsf{a}}_2, \dot{r}_1 \dot{\nabla}_r \dot{r}_2)$.

We conclude this section with two soundness lemmata. The proof for both can be found in [6]. The first one states that for each initial value and object approximation, all the history traces computed by $\mathcal{C}[\![\cdot]\!]^{\mathsf{a}}$ are of the form of $\langle \{\texttt{init}\}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \rangle \cdot r$, for some $\mathsf{v}^{\mathsf{a}} \in \mathsf{Val}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \in \mathsf{D}^{\mathsf{a}}$ and regular expression r. Intuitively, it means that the first interaction of the context with an object is the

invocation of `init` with some value and store configuration. This fact can be used to show that $[\![\mathtt{A}]\!]^{\mathsf{a}}$, as defined in the previous section, overapproximates the semantics of all the objects. Thus, that it is a sound class invariant.

Lemma 1 (Soundness of the class invariant). *Let* $\mathsf{d}^{\mathsf{a}}_0 \in \mathsf{D}^{\mathsf{a}}$, $\dot{\vartheta} \in \dot{\mathsf{O}}^{\mathsf{a}}$ *and* $\mathcal{C}[\![\mathtt{C}]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}}_0, \dot{\vartheta}, \lambda\vartheta.\varepsilon) = (\mathsf{d}^{\mathsf{a}}, \dot{r})$. *Then for all the abstract objects* ϑ *such that* $\dot{r}(\vartheta) \neq \varepsilon$*:*

(i) $\dot{r}(\vartheta) = \langle\{\mathtt{init}\}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}}\rangle \cdot r$*, for some* $\mathsf{v}^{\mathsf{a}} \in \mathsf{Val}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}} \in \mathsf{D}^{\mathsf{a}}$ *and* $r \in \mathsf{R}$*;*
(ii) $\mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(\dot{r}(\vartheta), \dot{\perp}_o) \sqsubseteq^{\mathsf{a}}_o [\![\mathtt{A}]\!]^{\mathsf{a}}$.

The next lemma shows that the history traces computed by $\mathcal{C}[\![\cdot]\!]^{\mathsf{a}}$ are an overapproximation of the history traces computed by $[\![\cdot]\!]^{\mathsf{a}}$. Thus, the soundness of $[\![\cdot]\!]^{\mathsf{a}}$ implies that the history traces are a sound approximation of the context.

Lemma 2 (Soundness of the history traces). *Let* $[\![\mathtt{C[o]}]\!]^{\mathsf{a}}(\perp^{\mathsf{a}}) = \mathsf{d}^{\mathsf{a}}$, $\alpha_v(\{\mathtt{o}\}) = \vartheta$ *and* $t = \langle\mathtt{init}, \mathsf{v}^{\mathsf{a}}, \mathsf{s}^{\mathsf{a}}\rangle \cdot \langle\mathtt{m}_1, \mathsf{v}^{\mathsf{a}}_1, \mathsf{s}^{\mathsf{a}}_1\rangle \ldots \langle\mathtt{m}_n, \mathsf{v}^{\mathsf{a}}_n, \mathsf{s}^{\mathsf{a}}_n\rangle$ *a sequence of method invocations of* ϑ *when the rules of Fig. 2 are used to derive* d^{a}*. Then* $\mathcal{C}[\![\mathtt{C[o]}]\!]^{\mathsf{a}}(\perp^{\mathsf{a}}, \lambda\vartheta.[\![\mathtt{A}]\!]^{\mathsf{a}}, \lambda\vartheta.\varepsilon) = (\mathsf{d}'^{\mathsf{a}}, \dot{r}')$ *is such that* $\mathsf{d}^{\mathsf{a}} \sqsubseteq^{\mathsf{a}} \mathsf{d}'^{\mathsf{a}}$ *and* $\mathcal{L}(t) \subseteq \mathcal{L}(\dot{r}'(\vartheta))$.

4.4 Putting It All Together

In this section we show how to combine the two abstract semantic functions $\mathcal{O}[\![\cdot]\!]^{\mathsf{a}}$ and $\mathcal{C}[\![\cdot]\!]^{\mathsf{a}}$ in order to obtain a separate compositional analysis of $\mathtt{C[o]}$. The functions $\mathcal{O}[\![\cdot]\!]^{\mathsf{a}}$ and $\mathcal{C}[\![\cdot]\!]^{\mathsf{a}}$ are mutually related. The first one takes as input an approximation of the context and it returns an approximation of the object semantics. The second one takes as input an approximation of the objects. It returns an abstract state and, for each abstract object ϑ, an approximation of the context that interacts with ϑ. Then it is natural to handle this mutual dependence with a fixpoint operator.

Nevertheless, before doing it we need to define formally the function $\Theta \in [\dot{\mathsf{R}} \to \dot{\mathsf{O}}^{\mathsf{a}}]$, that maps an interaction history $\dot{r}$ to a function $\dot{\vartheta}$ from abstract objects to their approximation. First we consider the set of the abstract objects that interact with the context, i.e. the abstract addresses whom interaction history is non-empty: $I = \{\vartheta \mid \dot{r}(\vartheta) \neq \varepsilon\}$. Next, we define a function that maps elements of I to their abstract semantics and the others to the class invariant $[\![\mathtt{A}]\!]^{\mathsf{a}}$:

$$\dot{\vartheta}_{\dot{r}} = \lambda\vartheta. \begin{cases} \mathcal{O}[\![\vartheta]\!]^{\mathsf{a}}(\dot{r}(\vartheta), \perp^{\mathsf{a}}_o) & \text{if } \vartheta \in I \\ [\![\mathtt{A}]\!]^{\mathsf{a}} & \text{otherwise.} \end{cases} \tag{1}$$

Moreover, we require that the more precise the abstract object, the more precise its abstract semantics. Therefore we perform the downward closure of $\dot{\vartheta}_{\dot{r}}$, to make it monotonic. Finally, the object abstractions function in a context $\dot{r}$, $\Theta \in [\dot{\mathsf{R}} \to \dot{\mathsf{O}}^{\mathsf{a}}]$, is defined as $\Theta(\dot{r}) = \lambda\vartheta.\, \sqcap^{\mathsf{a}}_o\{\dot{\vartheta}_{\dot{r}}(\vartheta') \mid \vartheta' \in \mathsf{Addr}^{\mathsf{a}} \text{ and } \vartheta \sqsubseteq^{\mathsf{a}}_v \vartheta'\}$. The function Θ is well-defined as $\mathsf{Addr}^{\mathsf{a}}$ is a sublattice of $\mathsf{Val}^{\mathsf{a}}$ and the monotonicity of $\Theta(\dot{r})$ is a direct consequence of the definition.

Using the above definition and defining $\Gamma(\dot{\vartheta}) = \mathcal{C}[\![\mathtt{C}]\!]^{\mathsf{a}}(\bot^{\mathsf{a}}, \dot{\vartheta}, \lambda\vartheta.\varepsilon)$, it is now possible to formally state the interdependence between the context and the objects semantics as follows:

$$\begin{aligned} \dot{\vartheta} &= \Theta(\dot{r}) \\ (\mathsf{d}^{\mathsf{a}}, \dot{r}) &= \Gamma(\dot{\vartheta}). \end{aligned} \tag{2}$$

A solution to the recursive equation (2) can be found with the standard iterative techniques. Nevertheless, our goal is to parallelize the iterative computation of Θ and Γ, in order to speed up the whole analysis. Therefore, we start the iterations by considering a worst-case approximation for $\dot{r}$ and $\dot{\vartheta}$: $\dot{r}_0 = \lambda\vartheta.r_\top$ and $\dot{\vartheta}_0 = \lambda\vartheta.\top^{\mathsf{a}}_o$. In other words, we assume an unknown context when computing the abstract object semantics and an unknown semantics when analyzing the context. Then we obtain $\dot{\vartheta}_1 = \Theta(\dot{r}_0)$ and $(\mathsf{d}^{\mathsf{a}}_1, \dot{r}_1) = \Gamma(\dot{\vartheta}_0)$.

As we consider the worst-case approximation for the objects semantics, the abstract state $\mathsf{d}^{\mathsf{a}}_1$ is an upper approximation of $[\![\mathtt{C}]\!]^{\mathsf{a}}(\bot^{\mathsf{a}})$. Furthermore, it is easy to see that $\dot{r}_1 \dot{\sqsubseteq}_r \dot{r}_0$ and $\dot{\vartheta}_1 \dot{\sqsubseteq}_o \dot{\vartheta}_0$. Roughly speaking, this means that after one iteration we have a better approximation of the context and the object semantics. As a consequence, if we compute $\dot{\vartheta}_2 = \Theta(\dot{r}_1)$ and $(\mathsf{d}^{\mathsf{a}}_2, \dot{r}_2) = \Gamma(\dot{\vartheta}_1)$, we obtain a better approximation for the abstract state, the semantics of the objects and that of the context. This process can be iterated, so that at step $i+1$ we have:

$$\begin{aligned} \dot{\vartheta}_{i+1} &= \Theta(\dot{r}_i) \\ (\mathsf{d}^{\mathsf{a}}_{i+1}, \dot{r}_{i+1}) &= \Gamma(\dot{\vartheta}_i). \end{aligned} \tag{3}$$

The next theorem synthesizes what has been said so far. It states that the iterations of (3) form a decreasing chain and that at each iteration step $\mathsf{d}^{\mathsf{a}}_{i+1}$ is an sound approximation of the monolithic abstract semantics. Hence, of the concrete semantics:

Theorem 2 (Soundness). *Let* $\mathtt{C}$ *be a context. Then* $\forall i \geq 0$.

(i) $\mathsf{d}^{\mathsf{a}}_{i+1} \sqsubseteq^{\mathsf{a}} \mathsf{d}^{\mathsf{a}}_i$, $\dot{r}_{i+1} \dot{\sqsubseteq}_r \dot{r}_i$ *and* $\dot{\vartheta}_{i+1} \dot{\sqsubseteq}_o \dot{\vartheta}_i$.
(ii) $[\![\mathtt{C}]\!]^{\mathsf{a}}(\bot^{\mathsf{a}}) \sqsubseteq^{\mathsf{a}} \mathsf{d}^{\mathsf{a}}_i$.

Roughly speaking the first point of the theorem states that the more the iterations the more precise the result of the analysis. On the other hand, the second point states that the abstract states are all above the result of the monolithic abstract semantics. As a consequence it is possible to stop the iterations at a step $\underline{i}$, the resulting abstract state $\mathsf{d}^{\mathsf{a}}_{\underline{i}}$ being a sound approximation of the concrete semantics.

An analysis based on (3) has several advantages. First, it is possible to use the asynchronous iterations with memory [1] in order to parallelize the analysis of the context and the objects. Intuitively, this is a consequence of the fact that at each iteration, the result of Θ and Γ depends just on the result of the previous iteration. Furthermore, Θ computes the abstract semantics for several, independent, abstract objects (cf. (1)). Therefore, even the effective implementation of

```
o1 = new A(5, 10);
o2 = new A(3, 10);
while ... do
  if o1.get_y() + o2.get_y() ≥ 0 then
    o1.addA(5); o1.addB(3);
  else
    o2.addA(7); o2.addA(1);
{ assert(Prop) }
```

(a) The context

```
F : {a, b, y}

init(a0, c0) : a = a0; b = c0 − a0; y = 0

addA(x) : a = a + x; b = b − x; y = y + 1
addB(x) : a = a − x; b = b + x; y = y − 1
get_y() : return y
```

(b) The class A

Fig. 6. Example of a context and a class

Θ may take advantage of a further parallelization. Finally the fact that each iteration is a sound approximation allows a fine tuning of the trade-off precision/cost. In particular, we can stop the iterations as soon as the desired degree of precision is reached.

Example 1. As an example, we can consider the context and the class A in Fig. 6, where Prop is the property: $(\mathsf{o}_1.a + \mathsf{o}_1.b) - (\mathsf{o}_2.a + \mathsf{o}_2.b) + (\mathsf{o}_1.y + \mathsf{o}_2.y) \geq 0$.

We are interested in proving that the assert condition is never violated. In order to do it, we instantiate the abstract domain D^a with Polyhedra [3], and we consider the two abstract objects ϑ_1 and ϑ_2 corresponding respectively to o_1 and o_2. According to the iteration schema (3), the first step approximates the objects semantics with the class invariant: $\Theta(\dot{r}_0) = \lambda\vartheta.\langle \mathsf{i}^\mathsf{a}, \lambda\mathtt{m}.\mathsf{p}^\mathsf{a}(\mathtt{m})\rangle$. The object fields invariant is $\mathsf{i}^\mathsf{a} = \{\mathsf{a} + \mathsf{b} = \mathsf{c}_0\}$ and the method postconditions are:

$$\mathsf{p}^\mathsf{a} = \begin{cases} \mathtt{init} \mapsto \langle \bot_v^\mathsf{a}, \mathsf{i}^\mathsf{a} \cup \{\mathsf{y} = 0\}\rangle \\ \mathtt{addA} \mapsto \langle \bot_v^\mathsf{a}, \mathsf{i}^\mathsf{a} \cup \{\mathsf{y} = \mathsf{y} + 1\}\rangle \\ \mathtt{addB} \mapsto \langle \bot_v^\mathsf{a}, \mathsf{i}^\mathsf{a} \cup \{\mathsf{y} = \mathsf{y} - 1\}\rangle \\ \mathtt{get\_y} \mapsto \langle \top_v^\mathsf{a}, \mathsf{i}^\mathsf{a}\rangle. \end{cases}$$

On the other hand, as far as the context analysis is concerned, we have $(\emptyset, \dot{r}_1) = \Gamma(\dot{\vartheta}_0)$, where $\dot{r}_1$ is the interaction history below. For lack of space we simplify the structure of the interaction history by omitting the abstract state. Nevertheless, in the example this is not problematic, as the objects do not expose the internal state.

$$\dot{r}_1 = \begin{cases} \vartheta_1 \mapsto \langle\{\mathtt{init}\}, (5, 10)\rangle \cdot (\langle\{\mathtt{get\_y}\}, \emptyset\rangle \cdot \langle\{\mathtt{addA}\}, 5\rangle \cdot \langle\{\mathtt{addB}\}, 3\rangle)^* \\ \vartheta_2 \mapsto \langle\{\mathtt{init}\}, (7, 10)\rangle \cdot (\langle\{\mathtt{get\_y}\}, \emptyset\rangle \cdot \langle\{\mathtt{addA}\}, 7\rangle \cdot \langle\{\mathtt{addA}\}, 1\rangle)^*. \end{cases}$$

The result of the next iteration, $\Gamma(\dot{\vartheta}_1)$, is still too imprecise for verifying the assertion, as the object fields invariant i^a implies that $(\mathsf{o}_1.a + \mathsf{o}_1.b) - (\mathsf{o}_2.a + \mathsf{o}_2.b) = 0$, but nothing can be said about $\mathsf{o}_1.y + \mathsf{o}_2.y$. Nevertheless, the analysis of the object semantics under the context $\dot{r}_1$ results in a more precise approximation of the objects semantics. In particular, we obtain for the first object the field invariant $\mathsf{i}_1^\mathsf{a} = \mathsf{i}^\mathsf{a} \cup \{0 \leq \mathsf{y} \leq 1\}$ and for the latter $\mathsf{i}_2^\mathsf{a} = \mathsf{i}^\mathsf{a} \cup \{\mathsf{y} \geq 0\}$. As a consequence, a

further iteration is enough to infer that the condition `Prop` is verified. From Th. 2 it follows that the result is sound, even if it is not the most precise one. In fact it is easy to see that a further iteration gives a more precise result, proving that the `else` branch in the conditional is never taken. Therefore that $\mathtt{o}_2.\mathtt{y}$ is identically equal to zero.

5 Conclusions and Future Work

In this work we introduced a separate compositional analysis and we proved it correct for a small yet realistic object-oriented language. In particular we presented an iteration schema for the computation of the abstract semantics that approximates it from above. The central idea for the parallelization is the use of a domain of regular expressions to encode the interactions between the context and the objects.

In future work we plan to study the practical effectiveness of the presented technique, for example with regard to memory consumption. Moreover, it would be interesting to study how many iterations are needed in order to reach an acceptable degree of precisions. As far as the theoretical point of view is concerned, a straightforward extension of this work is a direct handling of inheritance. Nevertheless, in our opinion the combination of the present work with modular techniques for the handling of inheritance presents some more challenges that deserve to be explored [5].

Acknowledgments. We would like to thank R. Cousot, J. Feret, C. Hymans, A. Miné and X. Rival for their comments.

References

1. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical Report R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, 1977.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. ACM Press, 1977.
3. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*. ACM Press, 1978.
4. F. Logozzo. Class-level modular analysis for object oriented languages. In *SAS'03*, volume 2694 of *LNCS*. Springer-Verlag, 2003.
5. F. Logozzo. Automatic inference of class invariants. In *VMCAI'04*, volume 2937 of *LNCS*. Springer-Verlag, 2004.
6. F. Logozzo. *Modular Static Analysis of Object Oriented Languages*. PhD thesis, École Polytechnique, France, 2004. To appear.
7. I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of Java programs. In *ECOOP'01*, volume 2072 of *LNCS*. Springer-Verlag, 2001.
8. A. Rountev, A. Milanova, and B.G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *ICSE'03*. IEEE Press, 2003.