Class Analysis of Object-Oriented Programs through Abstract Interpretation

Thomas Jensen and Fausto Spoto

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{jensen,spoto}@irisa.fr
Ph.: +33 0299847232 Fax: +33 0299842590

Abstract We use abstract interpretation to define a uniform formalism for presenting and comparing class analyses for object-oriented languages. We consider three domains for class analysis derived from three techniques present in the literature, viz., rapid type analysis, a simple dataflow analysis and the constraint technique of Palsberg and Schwartzbach. We obtain three static analyses which are provably correct and whose abstract operations are provably optimal. Moreover, we prove that our formalisation of the analysis of Palsberg and Schwartzbach is more precise than that of the dataflow analysis.

Keywords: *Abstract interpretation, class analysis, object-oriented languages, domain theory, semantics.*

1 Introduction

Class analysis, one of the most important analyses for object-oriented languages, computes the set of classes that an expression can have at run-time [1, 2, 10–12]. This information has several uses. It provides *type safety* by guaranteeing that methods are only invoked on objects that implement such a method and in certain cases it allows to optimise virtual method invocations into direct calls to the code implementing the method. Furthermore, it is used for building a precise call graph for a program which in turn enables other analyses. A variety of class analyses have been proposed[1], often using different analysis frameworks. This makes it difficult to compare the analyses w.r.t. their precision. Furthermore, the complex question of proving the analyses correct is not always addressed.

In [7], Cousot showed how *abstract interpretation* [8, 9] could organise eleven different type systems for functional languages into a lattice that allows to state their correctness and establish their relative precision. In this paper we apply abstract interpretation to classify and prove correct class analyses for object-oriented languages. Abstract interpretation is a technique for the systematic construction of semantics and semantics-based static analyses for programming languages. Given a concrete semantics over a concrete domain and an *abstraction*

[1] Sometimes, other names are being used: receiver class analysis, type analysis, ...

function from the concrete to an abstract semantic domain, abstract interpretation shows how to define an abstract semantics over the abstract domain such that the result of the analysis is provably *correct* w.r.t. the concrete one. The abstraction function can be seen as a specification of the program property of interest, and the abstract semantics as an analyser for that property. Furthermore, since abstract interpretation is a framework for relating semantic definitions, it can be used to compare the relative precision of two analyses.

Contributions. We have developed a framework for the static analysis of object-oriented programs. It gives semantics to a simple object-oriented language by specifying an algebra of states. The operations of this algebra are reminiscent of Java bytecode operations. Different static analyses are obtained by abstract interpretation of these operations. This approach allows to make the following three contributions:

The first contribution consists in defining three abstractions of the above algebra, namely, three domains for class analysis which use the same abstract information as [2], [10] and [11], respectively. In this way we obtain formal reconstructions of these analyses in a common framework.

The second contribution is the use of abstract interpretation to derive optimal (and hence correct) abstract operations for our three domains. Note that only the analysis in [11] has been provided with an (informal) correctness proof.

As a third contribution, we show that our reconstruction of the analysis in [11] is *provably* more precise than that of the analysis in [10].

We emphasise that we are not at all questioning the correctness of the original analyses. Nor are we aiming at defining new, more powerful class analyses (although the present framework could be used for this). The purpose of this work is to set up a framework for studying the common structure of class analyses, and to facilitate their comparison and proof of correctness by means of abstract interpretation.

Plan of the paper. Section 2 presents notation and preliminaries. Section 3 shows our framework for the analysis of object-oriented languages, the states and their operations. Sections 4, 5 and 6 define the analyses in [2], [10] and [11], respectively, as abstract interpretations of the sets of states of Section 3. Section 7 compares the precision of our three analyses. Section 8 concludes. For reasons of space, proofs have been usually omitted.

2 Preliminaries

We require basic knowledge of ordered sets, denotational semantics and object-oriented languages. We recall some notions of abstract interpretation [8, 9].

Let $(C, \leq)$ and $(A, \preceq)$ be two posets (the concrete and the abstract domain). A *Galois connection* is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that for each $x \in C$ we have $x \leq (\gamma \circ \alpha)(x)$, and for each $y \in A$ we have $(\alpha \circ \gamma)(y) \preceq y$. A *Galois insertion* is a Galois connection where $\alpha \circ \gamma$ is the identity map. This means that the abstract domain does not contain *useless* elements. The map α (γ) is called the *abstraction* (*concretisation*) function. The abstraction map uniquely determines the concretisation map and vice versa.

Let $f : C^n \to C$ be a concrete operator. We say that $\tilde{f} : A^n \to A$ is *correct* w.r.t. f if and only if for all $y_1, \dots, y_n \in A$ we have $\alpha(f(\gamma(y_1), \dots, \gamma(y_n))) \preceq \tilde{f}(y_1, \dots, y_n)$. For each operator f, there exists an *optimal* (most precise) correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \dots, y_n) = \alpha(f(\gamma(y_1), \dots, \gamma(y_n)))$. This means that $\tilde{f}$ is the most precise abstraction of f which can be defined on A.

In the theory of abstract interpretation, the *semantics* of a program is the fixpoint of a continuous operator $f : C \mapsto C$, where C is the computational domain [6]. Its *collecting version* [8] works over *properties* of C, i.e., over $\wp(C)$ and is the fixpoint of the powerset extension of f. If f is defined through smaller operations, their powerset extensions *and* $\cup$ induce the extension of f. The $\cup$ operation merges the semantics of the branches of a conditional. Note that the powerset extension of a predicate over C is a constant of $\wp(C)$. In Props. 2, 4 and 6, we compute the optimal abstractions of the powerset extension of every small operation g above as $\alpha \circ g \circ \gamma$, consistently with the previous paragraph.

Upper closure operators (*uco*'s) are monotonic, extensive and idempotent maps. They are isomorphic to Galois insertions [9]. The abstract domain induced by an uco ρ over the concrete domain L (for our purposes, the semantic domain $\wp(C)$ of a collecting semantics) is its image $\rho(L)$, i.e., the set of its fixpoints. This image is a *Moore family* of L, i.e., a non-empty meet-closed subset of L. Conversely, any Moore family of L is the image of some uco on L. Thus, in order to define a Galois insertion on L, i.e., an abstraction of L, we can equivalently define a Moore family on L, whose induced uco is the abstraction map. In Props. 1, 3 and 5 we use this technique to prove that we deal with Galois insertions.

A definition like $S = \langle a, b\rangle$, with a and b meta-variables, defines the selectors $s.a$ and $s.b$ for $s \in S$. For instance, Def. 2 defines $o.\kappa$ and $o.\phi$, for $o \in Obj$.

We denote by $[v_1 \mapsto t_1, \dots, v_n \mapsto t_n]$ a function f whose domain is $\{v_1, \dots, v_n\}$ and such that $f(v_i) = t_i$ for $i = 1, \dots, n$. Its update is $f[d_1/w_1, \dots, d_m/w_m]$. Its domain and codomain are $\mathsf{dom}(f)$ and $\mathsf{codom}(f)$, respectively. By $f|_s$ we denote the restriction of f to $s \subseteq \mathsf{dom}(f)$, by $f|_{-s}$ its restriction to $\mathsf{dom}(f) \setminus s$. A total function is denoted by $\mapsto$. A partial function by $\to$.

If $\langle D, \leq\rangle$ is a poset and $d \in D$, we define $\downarrow(d) = \{d' \in D \mid d' \leq d\}$.

3 Semantic domains and operations

We describe here the semantic domain of program states and their operations. A transition trace semantics (where the state is a stack of activation frames) can be defined either in the style of Bertelsen's small-step operational semantics [3] or in that of Börger and Schulte' Abstract State Machines [4]. Our set of operations is not sufficiently complete to give semantics for a concrete language like Java, but it is representative of the set of operations that would be required for such a task. Furthermore, once the correctness of the abstraction of these operations is proved, that of the whole abstract interpretation follows by fairly standard means [8,9].

For simplicity, integers and booleans are the only basic types. There are no static fields or methods. Methods are uniquely identified by a *Method_ref* (for instance, their fully qualified name, like in Java), and classes by a *Class_ref*. We define $Type = \{int, bool\} \cup Class\_ref$, $Int = \mathbb{Z}$ and $Bool = \{true, false\}$.

In the following, we assume that the class structure of a program is specified by some $d \in Decl$, $p \in Pars$ and $n \in Name$ (see below), which give the classes, the parameters of the methods and their names, respectively, together with a subtype relation $\leq$ on $Type$. For simplicity, we do not allow basic types to have subtypes. By *instance variables* we refer to fields. A *program variable* is a local variable, a parameter of a method or `this`. By *class variables* we denote the variables and fields which can reference objects.

Definition 1. *Let Id be an infinite set of* identifiers, *with* $\mathtt{this} \in Id$. *We define*

$$Typing = \left\{ \tau : Id \to Type \;\middle|\; \begin{array}{l} \mathsf{dom}(\tau) \textit{ is finite} \\ \textit{if } \mathtt{this} \in \mathsf{dom}(\tau) \textit{ then } \tau(\mathtt{this}) \in Class\_ref \end{array} \right\}$$

$$Class = Typing \times (Id \to Method\_ref) \qquad Decl = (Class\_ref \mapsto Class)$$

$$Pars = (Method\_ref \mapsto \mathsf{seq}(Id) \times Typing) \quad Name = (Method\_ref \mapsto Id)\ .$$

If $cl \in Class$, we name its components as $cl = \langle \tau, \nu \rangle$. If $p \in Pars$, $n \in Name$ and $mr \in Method\_ref$ then $p(mr) = \langle s, \tau \rangle$, where $\mathtt{this}, n(mr) \in s$ and $\mathsf{dom}(\tau) = s$.

Typings map variables to types, but `this` can be bound only to objects. The indirect definition of classes through $Decl$ allows to define two classes with the same structure. This structure is a pair whose first component is the typing of the fields defined or inherited by the class, and the second the table of the methods defined or inherited by the class. The map $Pars$ binds every method reference to the typing (its signature) and its list of parameters. This list, whose elements are the domain of the typing, is useful since it yields their order in the definition of the method. The map $Name$ binds a method reference to its name.

Example 1. Assume that we have two classes `a` and `b`, subclass of `a`, and that `a` (and, hence, `b`) has a field `n` of class `a`. We model this situation as $Class\_ref = \{\kappa_a, \kappa_b\}$, $Decl = [\kappa_a \mapsto a, \kappa_b \mapsto b]$, $a = b = \langle [\mathtt{n} \mapsto \kappa_a], [] \rangle$ and $\kappa_b \leq \kappa_a$. Note how a and b have the same structure but have different class references.

Frames yield values to the variables addressable at a given program point. The special variable `this` cannot be unbound. Memories map locations to objects. Objects contain their *Class_ref* and the frame of their instance variables.

Definition 2. *Let Loc be an infinite set of* locations. *We define $Value = Int + Bool + Loc + \{nil\}$ and, for every $\tau \in Typing$,*

$$Frame_\tau = \left\{ \phi \;\middle|\; \begin{array}{l} \phi \in \mathsf{dom}(\tau) \mapsto Value \textit{ and for every } v \in \mathsf{dom}(\tau) \\ \textit{if } \tau(v) = int \textit{ then } \phi(v) \in Int \\ \textit{if } \tau(v) = bool \textit{ then } \phi(v) \in Bool \\ \textit{if } \tau(v) \in Class\_ref \textit{ then } \phi(v) \in \{nil\} \cup Loc \\ \textit{if } \mathtt{this} \in \mathsf{dom}(\tau) \textit{ then } \phi(\mathtt{this}) \in Loc \end{array} \right\}$$

$$
\begin{aligned}
\mathsf{nop}_\tau &: State_\tau \mapsto State_\tau \\
\mathsf{get\_int}^i_\tau &: State_\tau \mapsto State_{\tau[int/res]} \qquad \text{with } res \notin \mathsf{dom}(\tau),\ i \in Int \\
\mathsf{get\_nil}^{cr}_\tau &: State_\tau \mapsto State_{\tau[cr/res]} \qquad \text{with } res \notin \mathsf{dom}(\tau),\ cr \in \mathit{Class\_ref} \\
\mathsf{get\_bool}^b_\tau &: State_\tau \mapsto State_{\tau[bool/res]} \qquad \text{with } res \notin \mathsf{dom}(\tau),\ b \in Bool \\
\mathsf{get\_var}^v_\tau &: State_\tau \mapsto State_{\tau[\tau(v)/res]} \qquad \text{with } v \in \mathsf{dom}(\tau),\ res \notin \mathsf{dom}(\tau) \\
\mathsf{get\_field}^f_\tau &: State_\tau \to State_{\tau[d(\tau(res)).\tau(f)/res]} \\
&\quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathit{Class\_ref},\ f \in \mathsf{dom}(d(\tau(res)).\tau) \\
\mathsf{put\_var}^v_\tau &: State_\tau \mapsto State_{\tau|_{-res}} \quad \text{with } res \in \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau),\ v \neq res,\ \tau(res) \leq \tau(v) \\
\mathsf{put\_field}^f_{\tau,\tau'} &: State_\tau \mapsto (State_{\tau'} \to State_{\tau|_{-res}})) \\
&\quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathit{Class\_ref},\ f \in \mathsf{dom}(d(\tau(res)).\tau), \\
&\quad \tau' = \tau[t/res] \text{ with } t \leq d(\tau(res)).\tau(f) \\
=_\tau &: State_\tau \mapsto (State_\tau \mapsto State_{\tau[bool/res]}) \qquad \text{with } res \in \mathsf{dom}(\tau) \\
\mathsf{scope}^{mr,v_1,\ldots,v_{\#p(mr).s-2}}_\tau &: State_\tau \mapsto State_{p(mr).\tau|_{-n(mr)}} \\
&\quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathit{Class\_ref}, \\
&\quad p(mr).s \setminus \{n(mr), this\} = \langle \iota_1, \ldots, \iota_n \rangle,\ \tau(res) \leq p(mr).\tau(this), \\
&\quad v_i \in \mathsf{dom}(\tau) \text{ and } \tau(v_i) \leq p(mr).\tau(\iota_i) \text{ for every } i = 1, \ldots, \#p(mr).s - 2 \\
\mathsf{unscope}^{mr}_\tau &: State_\tau \mapsto (State_{p(mr).\tau|_{n(mr)}} \to State_{\tau[p(mr).\tau(n(mr))/res]}) \quad \text{with } res \notin \mathsf{dom}(\tau) \\
\mathsf{restrict}^{vs}_\tau &: State_\tau \mapsto State_{\tau|_{-vs}} \qquad \text{with } vs \subseteq \mathsf{dom}(\tau) \\
\mathsf{expand}^{v:t}_\tau &: State_\tau \mapsto State_{\tau[t/v]} \qquad \text{with } v \notin \mathsf{dom}(\tau),\ t \in Type \\
\mathsf{lookup}^{id,mr}_\tau &\subseteq \wp(State_\tau) \\
&\quad \text{with } res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathit{Class\_ref},\ id \in \mathsf{dom}(d(\tau(res)).\nu),\ mr \in \mathit{Method\_ref} \\
\mathsf{new}^{cr}_\tau &: State_\tau \mapsto State_{\tau[cr/res]} \qquad \text{with } res \notin \mathsf{dom}(\tau),\ cr \in \mathit{Class\_ref} \\
\mathsf{is\_true}_\tau, \mathsf{is\_false} &\subseteq \wp(State_\tau) \qquad \text{with } \tau(res) = bool\ .
\end{aligned}
$$

Figure1. The signatures of the operations over the states.

$$
Memory = \left\{ \begin{array}{l} \mu \in Loc \to Obj \\ \mathsf{dom}(\mu) \ \textit{is finite} \end{array} \right\} \qquad Obj = \left\{ \langle \kappa, \phi \rangle \,\middle|\, \begin{array}{l} \kappa \in \mathit{Class\_ref} \\ \phi \in Frame_{d(\kappa).\tau} \end{array} \right\} .
$$

Given $\mu_1, \mu_2 \in Memory$, we say that μ_2 is an update *of μ_1, written $\mu_1 \lhd \mu_2$, if $\mathsf{dom}(\mu_1) \subseteq \mathsf{dom}(\mu_2)$ and for every $l \in \mathsf{dom}(\mu_1)$ we have $\mu_1(l).\kappa = \mu_2(l).\kappa$.*

Def. 2 states that the types of the variables in a frame are consistent with its typing, but class variables are just required to be bound to *nil* or a location (only a location for `this`). In order to guarantee that they contain objects of a class compatible with the typing, we define states, i.e., pairs of frame and memory.

Definition 3. *Let $\tau \in Typing$. Recall that $\leq$ is the subtype relation. We say that $\phi \in Frame_\tau$ is τ-*correct *in a memory μ, written $\phi \propto_\tau \mu$, if it binds the class variables in its domain to objects of classes compatible with τ. Namely, $\phi \propto_\tau \mu$ if and only if for every $v \in \mathsf{dom}(\phi)$ such that $\phi(v) \in Loc$ we have $\phi(v) \in \mathsf{dom}(\mu)$*

$$\mathsf{nop}_\tau(\langle\phi,\mu\rangle)=\langle\phi,\mu\rangle \qquad \mathsf{get\_int}^i_\tau(\langle\phi,\mu\rangle)=\langle\phi[i/res],\mu\rangle$$

$$\mathsf{get\_nil}^{cr}_\tau(\langle\phi,\mu\rangle)=\langle\phi[nil/res],\mu\rangle \qquad \mathsf{get\_bool}^b_\tau(\langle\phi,\mu\rangle)=\langle\phi[b/res],\mu\rangle$$

$$\mathsf{get\_var}^v_\tau(\langle\phi,\mu\rangle)=\langle\phi[\phi(v)/res],\mu\rangle \qquad \mathsf{put\_var}^v_\tau(\langle\phi,\mu\rangle)=\langle\phi[\phi(res)/v]|_{-res},\mu\rangle$$

$$\mathsf{get\_field}^f_\tau(\langle\phi',\mu\rangle)=\begin{cases}\langle\phi'[(\mu(\phi'(res)).\phi)(f)/res],\mu\rangle & \text{if } \phi'(res)\neq nil\\ undefined & \text{otherwise}\end{cases}$$

$$\mathsf{put\_field}^f_{\tau,\tau'}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle)=\begin{cases}\langle\phi_2|_{-res},\mu_2[\langle\mu_2(l).\kappa,\mu_2(l).\phi[\phi_2(res)/f]\rangle/l]\rangle\\ \qquad \text{if } \mu_1\lhd\mu_2 \text{ and } (l=\phi_1(res))\neq nil\\ undefined\\ \qquad \text{otherwise}\end{cases}$$

$$=_\tau(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle)=\begin{cases}\langle\phi_2[true/res],\mu_2\rangle\\ \qquad \text{if } \mu_1\lhd\mu_2 \text{ and } \phi_1(res)=\phi_2(res)\\ \langle\phi_2[false/res],\mu_2\rangle\\ \qquad \text{if } \mu_1\lhd\mu_2 \text{ and } \phi_1(res)\neq\phi_2(res)\\ undefined\\ \qquad \text{otherwise}\end{cases}$$

$$\mathsf{scope}^{mr,v_1,\dots,v_n}_\tau(\langle\phi,\mu\rangle)=\langle[\iota_1\mapsto\phi(v_1),\dots,\iota_n\mapsto\phi(v_n),\mathtt{this}\mapsto\phi(res)],\mu\rangle$$
$$\text{where } \langle\iota_1,\dots,\iota_n\rangle=p(mr).s\setminus\{n(mr),\mathtt{this}\}$$

$$\mathsf{unscope}^{mr}_\tau(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle)=\begin{cases}\langle\phi_1[\phi_2(n(mr))/res],\mu_2\rangle & \text{if } \mu_1\lhd\mu_2\\ undefined & \text{otherwise}\end{cases}$$

$$\mathsf{restrict}^{vs}_\tau(\langle\phi,\mu\rangle)=\langle\phi|_{-vs},\mu\rangle \qquad \mathsf{expand}^{v:t}_\tau(\langle\phi,\mu\rangle)=\langle\phi[\mathsf{init}(t)/v],\mu\rangle$$

$$\mathsf{lookup}^{id,mr}_\tau(\langle\phi,\mu\rangle) \text{ if and only if } \phi(res)\neq nil \text{ and } d(\mu(\phi(res)).\kappa).\nu(id)=mr$$

$$\mathsf{new}^{cr}_\tau(\langle\phi,\mu\rangle)=\langle\phi[l/res],\mu[\langle cr,\mathsf{init}(d(cr).\tau)\rangle/l]\rangle \qquad l\in Loc \text{ fresh}$$

$$\mathsf{is\_true}_\tau(\langle\phi,\mu\rangle) \text{ if and only if } \phi(res)=true \qquad \mathsf{is\_false}_\tau(\langle\phi,\mu\rangle) \text{ if and only if } \phi(res)=false,$$

where $\mathsf{init}(int)=0$, $\mathsf{init}(bool)=false$, $\mathsf{init}(cr)=nil$ for $cr\in Class\_ref$ and $\mathsf{init}(\tau)=\lambda v\in\mathsf{dom}(\tau).\mathsf{init}(\tau(v))$ for $\tau\in Typing$.

Figure2. The operations over the states.

and $(\mu(\phi(v))).\kappa\leq\tau(v)$. *We define*

$$State_\tau=\left\{\langle\phi,\mu\rangle\;\middle|\;\begin{array}{l}\phi\in Frame_\tau,\ \mu\in Memory,\ \phi\propto_\tau\mu\\ and\ for\ every\ \langle\kappa,\phi'\rangle\in\mathsf{codom}(\mu)\ we\ have\ \phi'\propto_{d(\kappa).\tau}\mu\end{array}\right\}.$$

This set forms an algebraic structure with the operations whose signatures are given in Figure 1 and which are explicitly defined in Figure 2.

In these operations the variable *res* stands for the place where intermediate results are stored and retrieved. In a semantics for a stack-based language such as the Java bytecode, *res* would be the top of the operand stack. The binary operations of Figure 2 are undefined when the memory of the second argument is not an update of that of the first one, to guarantee that the $\propto_\tau$ relation between frame and memory of a state (Definition 3) is not broken by the operation.

Let τ be the typing of the variables addressable in a given program point. The nop operation does nothing. The get_int operation loads an integer into *res*. Similarly for get_nil and get_bool. The get_var operation fetches the value of a variable v and loads it into *res*. Then its result has typing $\tau[\tau(v)/res]$. The get_field operation fetches from res a non-nil reference to the object containing the field. The content of this field is loaded into res, whose type is updated with the declared type of the field. Note how this content is found. Since $\phi'(res)$ points to the object containing the field, $\mu(\phi'(res))$ is that object. Its fields are in $\mu(\phi'(res)).\phi$ and the field f is then $\mu(\phi'(res)).\phi(f)$. The put_var operation copies the content of res into v. The declared type of res must be a subtype of that of v. There is no resulting value. Thus, the variable res is removed from the typing of the result. In the put_field operation the res variable of the first argument points to the target object, while that of the second argument contains the new value for the field, whose declared type must be a subtype of that of the field. By using two states instead of an object and a value, we deal with just one semantic domain. In Fig. 2 we see its implementation. Since $l = \phi_1(res)$ points to the target object, it must not be nil and $\mu_2(l)$ is that object (since $\mu_1 \lhd \mu_2$). We write $\phi_2(res)$ in the variable f of the frame $\mu_2(l).\phi$ of that object. For every binary operation of the language there is a corresponding binary operation on states, like the shown case of $=$.

There are four operations (scope, unscope, restrict and expand) for modifying the structure of frames and states. The operations scope and unscope are used before and after a method call, respectively. The former creates a new frame in which the invoked method mr can execute. Its typing is $p(mr).\tau|_{-n(mr)}$ since the variable $n(mr)$, i.e., the name of the method, is not among its inputs. Note that $p(mr).\tau|_{-n(mr)}$ contains exactly the parameters of the method and the implicit `this` parameter. In this operation res points to the object over which the method is called and becomes the new `this` variable, which justifies the subtype check in Fig. 1. The first argument of the unscope operation is the state before the method call. The second is the state at the end of the execution of the method. The frame of this second state contains just a variable with the name of the method, and holds its result, like in Pascal procedures (in a complete operational semantics, the frames of these two states would be the top two elements of the stack). The operation restores the frame at the beginning of the execution of the method, by copying into res its result, and yields the memory at the end of its execution. The restrict operation removes some variables from a state and expand adds a new initialised variable to a state.

The lookup predicate models dynamic method lookup. It holds if by invoking the method id on the object referenced by res (which must not be nil) the method referenced by mr is selected. The expression $\mu(\phi(res))$ is the object on which the method is invoked. Then $\mu(\phi(res)).\kappa$ is its class reference and $d(\mu(\phi(res)).\kappa).\nu$ is the list of its methods. We check if by calling id we select the method mr by checking if $d(\mu(\phi(res)).\kappa).\nu(id) = mr$. The new operation creates a new initialised object and loads it into res. The is_true (is_false) predicate contains those states whose res variable contains $true$ ($false$).

Example 2. In the situation of Example 1, consider how the following sequence of operations change the state (read downwards).

$$\mathsf{nop}_\tau^{rta}(s) = (\mathsf{get\_int}_\tau^i)^{rta}(s) = (\mathsf{get\_nil}_\tau^{cr})^{rta}(s) = \mathsf{is\_true}_\tau^{rta}(s) = s$$
$$(\mathsf{get\_bool}_\tau^b)^{rta}(s) = (\mathsf{get\_var}_\tau^v)^{rta}(s) = (\mathsf{put\_var}_\tau^v)^{rta}(s) = \mathsf{is\_false}_\tau^{rta}(s) = s$$
$$(\mathsf{get\_field}_\tau^f)^{rta}(s) = \begin{cases} \emptyset & \text{if } s \cap \downarrow\tau(res) = \emptyset \\ s & \text{otherwise} \end{cases}$$
$$(\mathsf{put\_field}_{\tau,\tau'}^f)^{rta}(s_1)(s_2) = \begin{cases} \emptyset & \text{if } s_1 \cap s_2 \cap \downarrow\tau(res) = \emptyset \\ s_2 & \text{otherwise} \end{cases}$$
$$=_\tau^{rta}(s_1)(s_2) = s_2 \qquad (\mathsf{scope}_\tau^{mr,v_1,\dots,v_n})^{rta}(s) = s$$
$$(\mathsf{unscope}_\tau^{mr})^{rta}(s_1)(s_2) = s_2 \qquad (\mathsf{restrict}_\tau^{vs})^{rta}(s) = s$$
$$(\mathsf{expand}_\tau^{v:t})^{rta}(s) = s \qquad (\mathsf{new}_\tau^{cr})^{rta}(s) = s \cup \{cr\} \qquad \cup_\tau^{rta}(s_1)(s_2) = s_1 \cup s_2$$
$$(\mathsf{lookup}_\tau^{id,mr})^{rta}(s) = \begin{cases} s & \\ & \text{if there exists } \kappa \in s \cap \downarrow\tau(res) \\ & \text{such that } d(\kappa).\nu(id) = mr \\ \emptyset & \\ & \text{otherwise.} \end{cases}$$

Figure3. The instantiation over $State^{rta}$ of the operations of Figure 2 and of $\cup$.

Operation	State
	$\langle [v_1 \mapsto l_1], [l_1 \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ (initial state)
$\mathsf{expand}_{[v_1 \mapsto \kappa_a]}^{v_2:\kappa_a}$	$\langle [v_1 \mapsto l_1, v_2 \mapsto nil], [l_1 \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$
$\mathsf{new}_{[v_1,v_2 \mapsto \kappa_a]}^{\kappa_b}$	$\langle [v_1 \mapsto l_1, v_2 \mapsto nil, res \mapsto l_2], [l_1 \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle, l_2 \mapsto \langle \kappa_b, [\mathtt{n} \mapsto nil] \rangle] \rangle$
$\mathsf{get\_field}_{[v_1,v_2 \mapsto \kappa_a, res \mapsto \kappa_b]}^{\mathtt{n}}$	$\langle [v_1 \mapsto l_1, v_2 \mapsto nil, res \mapsto nil], [l_1 \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle, l_2 \mapsto \langle \kappa_b, [\mathtt{n} \mapsto nil] \rangle] \rangle$
$\mathsf{put\_var}_{[v_1,v_2,res \mapsto \kappa_a]}^{v_2}$	$\langle [v_1 \mapsto l_1, v_2 \mapsto nil], [l_1 \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle, l_2 \mapsto \langle \kappa_b, [\mathtt{n} \mapsto nil] \rangle] \rangle$

4 Rapid type analysis (rta-analysis)

Bacon and Sweeney defined [2] a simple and efficient *rapid type analysis* (rta) of little precision. They collect the set N of classes instantiated by a `new` command. The classes of an expression e are $(\downarrow D(e)) \cap N$, where $D(e)$ is the declared type of e. In terms of abstract interpretation, this technique amounts to defining an abstraction which collects the set of classes of the objects in the memory of the states. A priori, abstract interpretation can distinguish this set of classes in different program points. Then it results in a more precise analysis than that in [2].

Definition 4. *For $\tau \in Typing$, the abstract domain is $State_\tau^{rta} = \wp(Class\_ref)$ with the concretisation map $\gamma^{rta} : State_\tau^{rta} \mapsto \wp(State_\tau)$ such that (here, $o \in \mathsf{codom}(\mu)$ means that o is an object in the memory μ) $\gamma^{rta}(s) = \{\langle \phi, \mu \rangle \in State_\tau \mid$ for every $\langle \kappa, \phi' \rangle \in \mathsf{codom}(\mu)$ we have $\kappa \in s\}$.*

Proposition 1. *For $\tau \in Typing$, the set $\gamma^{rta}(State_\tau^{rta})$ is a Moore family of $\wp(State_\tau)$. Hence $State_\tau^{rta}$ is an abstract domain whose induced abstraction map is $\alpha^{rta} : \wp(State_\tau) \mapsto State_\tau^{rta}$ such that*

$$\alpha^{rta}(S) = \{\kappa \in Class\_ref \mid \langle \phi, \mu \rangle \in S \text{ and } \langle \kappa, \phi' \rangle \in \mathsf{codom}(\mu)\} \ .$$

Proof (Sketch). The non-empty set $\gamma^{rta}(State_\tau^{rta})$ is a Moore family since it is $\cap$-closed. Indeed, it can be shown that for every $\{s_i\}_{i\in\mathbb{N}} \subseteq State_\tau^{rta}$ we have $\cap_{i\in\mathbb{N}}\gamma_\tau^{rta}(s_i) = \gamma_\tau^{rta}(\cap_{i\in\mathbb{N}}s_i)$. The α^{rta} map is derived [8, 9] as $\alpha^{rta}(S) = \cap\{s \in State_\tau^{rta} \mid S \subseteq \gamma^{rta}(s)\} = \cap\{s \in \wp(Class\_ref) \mid \langle\phi,\mu\rangle \in S$ and $\langle\kappa,\phi'\rangle \in \mathsf{codom}(\mu)$ entails $\kappa \in s\} = \{\kappa \in Class\_ref \mid \langle\phi,\mu\rangle \in S$ and $\langle\kappa,\phi'\rangle \in \mathsf{codom}(\mu)\}$.

Proposition 2. *The optimal abstract counterparts of the powerset extension of the operations of Figure 2 and of* $\cup$ *are those given in Figure 3.*

Proof (Sketch). According to the general theory (Section 2), for the powerset extension g of every concrete operation in Figure 2 (and of $\cup$) we need to compute $\alpha^{rta}(g(\gamma^{rta}(s)))$, for every $s \in State_\tau^{rta}$. For the case $g = \mathsf{lookup}_\tau^{id,mr}$ we have $\alpha^{rta}(\mathsf{lookup}_\tau^{id,mr}(\{\langle\phi,\mu\rangle \in State_\tau \mid \langle\kappa,\phi'\rangle \in \mathsf{codom}(\mu)$ entails $\kappa \in s\})) = \alpha^{rta}(\{\langle\phi,\mu\rangle \in State_\tau \mid \phi(res) \neq nil,\ d(\mu(\phi(res)).\kappa).\nu(id) = mr$ and if $\langle\kappa,\phi'\rangle \in \mathsf{codom}(\mu)$ then $\kappa \in s\})$. If there is no $\kappa \in s \cap \downarrow\tau(res)$ with $d(\kappa).\nu(id) = mr$, there is no way of binding $\phi(res)$, the argument of α^{rta} is the empty set and $\alpha^{rta}(\mathsf{lookup}_\tau^{id,mr}(s)) = \emptyset$. If such a κ exists, given $\langle\phi,\mu\rangle \in State_\tau$ we can define $\langle\phi',\mu'\rangle \in State_\tau$ such that $\mu' = \mu[\langle\kappa, \mathsf{init}(d(\kappa).\tau)\rangle/l]$, with $l \in Loc$ fresh, and $\phi' = \phi[l/res]$. We have $\phi'(res) \neq nil$ and $d(\mu'(\phi'(res)).\kappa).\nu(id) = mr$. Since the contribution of $\langle\phi',\mu'\rangle$ to α^{rta} contains that of $\langle\phi,\mu\rangle$ (Prop. 1), the set above is $\alpha^{rta}(\{\langle\phi,\mu\rangle \in State_\tau \mid \langle\kappa,\phi'\rangle \in \mathsf{codom}(\mu)$ entails $\kappa \in s\}) = \alpha^{rta}(\gamma^{rta}(s)) = s$, where the last equality holds because $\alpha^{rta}\gamma^{rta}$ is the identity map.

The only operation in Figure 3 which enlarges the set of classes created during the execution of the program is new. The operations get_field and put_field check if a class compatible with the declared type of res has been instantiated, since otherwise res must be bound to nil, and the concrete operation fails. Similarly, lookup checks if it has been instantiated some class such that a call to the method id results in the execution of the method referenced by mr. Note how these operations allow the set of classes instantiated to shrink. This is a consequence of the use of abstract interpretation, which deals with different program points. It is a distinguishing feature w.r.t. the original technique in [2].

Example 3. We execute over $State_\tau^{rta}$ the same sequence of instructions of Ex. 2. The initial state is the abstraction (Prop. 1) of the initial concrete state.

Operation	State
	$\{\kappa_a\}$ (initial state)
$(\mathsf{expand}_{[v_1\mapsto\kappa_a]}^{v_2:\kappa_a})^{rta}$	$\{\kappa_a\}$
$(\mathsf{new}_{[v_1,v_2\mapsto\kappa_a]}^{\kappa_b})^{rta}$	$\{\kappa_a,\kappa_b\}$
$(\mathsf{get\_field}_{[v_1,v_2\mapsto\kappa_a,res\mapsto\kappa_b]}^{\mathtt{n}})^{rta}$	$\{\kappa_a,\kappa_b\}$
$(\mathsf{put\_var}_{[v_1,v_2,res\mapsto\kappa_a]}^{v_2})^{rta}$	$\{\kappa_a,\kappa_b\}$

The final abstract state says that v_1 and v_2, both of declared type a, can be bound to objects of class a or b, which is a correct but rather imprecise approximation of the concrete final state of Example 2.

5 Dataflow analysis (*df*-analysis)

In the dataflow analysis of [10] only program variables are analysed, while fields are approximated with the downwards closure $\downarrow(t)$ of their declared type t.

Compare this with Definition 2. An abstract value is a set of types which share a supertype. An abstract frame maps variables into abstract values consistent with the given typing. The special variable `this` cannot be unbound.

Definition 5. *Given $\tau \in Typing$, we define $Value^{df} = \{S \in \wp(Type) \mid S \subseteq \downarrow t$ for some $t \in Type\}$ and*

$$Frame^{df}_{\tau} = \left\{ \phi \;\middle|\; \begin{array}{l} \phi \in \mathsf{dom}(\tau) \mapsto Value^{df} \text{ and for every } v \in \mathsf{dom}(\tau) \\ \text{if } \tau(v) = int \text{ then } \phi(v) = \{int\} \\ \text{if } \tau(v) = bool \text{ then } \phi(v) = \{bool\} \\ \text{if } \tau(v) \in Class\_ref \text{ then } \phi(v) \subseteq \downarrow\tau(v) \\ \text{and if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ then } \phi(\mathtt{this}) \neq \emptyset \end{array} \right\}.$$

In $Frame^{df}_{\tau}$, integer and boolean variables are always mapped to $\{int\}$ and $\{bool\}$, respectively. We consider them just to simplify the presentation.

There is no abstract memory. Then abstract states are just abstract frames (compare with Definition 3). The special abstract state $\emptyset$ represents the empty set of concrete states and improves the precision of the analysis.

Definition 6. *For $\tau \in Typing$, the abstract domain is $State^{df}_{\tau} = \emptyset \cup Frame^{df}_{\tau}$.*

The relation $\approx$ says when an abstract value (i.e., a set of types) approximates a concrete value. The set $\{int\}$ approximates the integers, the set $\{bool\}$ approximates the booleans and a set of classes S approximates nil and all locations containing an object of a class in S.

Definition 7. *Let $\mu \in Memory$. We define $\approx_{\mu}$: $Value^{df} \times Value$ as the minimal relation such that $\{int\} \approx_{\mu} i$, with $i \in Int$, $\{bool\} \approx_{\mu} b$, with $b \in Bool$, $S \approx_{\mu} nil$, with $S \neq \{int\}$ and $S \neq \{bool\}$, and $S \approx_{\mu} l$ with $l \in Loc$ and $\mu(l).\kappa \in S$. This relation is pointwise extended to $\approx_{\mu}$: $Frame^{df}_{\tau} \times Frame_{\tau}$, for $\tau \in Typing$.*

The concretisation of an abstract state ϕ is the set of concrete states $\langle \phi', \mu \rangle$ whose frame ϕ' binds every variable $v \in \mathsf{dom}(\phi)$ to a concrete value $\phi'(v)$ approximated ($\approx_{\mu}$) by the abstract value $\phi(v)$, i.e., $\phi(v) \approx_{\mu} \phi'(v)$.

Definition 8. *For $\tau \in Typing$, we define the concretisation map $\gamma^{df} : State^{df}_{\tau} \mapsto \wp(State_{\tau})$ such that $\gamma^{df}(\emptyset) = \emptyset$ and $\gamma^{df}(\phi) = \{\langle \phi', \mu \rangle \in State_{\tau} \mid \phi \approx_{\mu} \phi'\}$ for every $\phi \in State^{df}_{\tau} \setminus \emptyset$.*

Proposition 3. *For $\tau \in Typing$, the set $\gamma^{df}(State^{df}_{\tau})$ is a Moore family of $\wp(State_{\tau})$. Hence $State^{df}_{\tau}$ is an abstract domain whose induced abstraction map is $\alpha^{df} : \wp(State_{\tau}) \mapsto State^{df}_{\tau}$ such that $\alpha^{df}(\emptyset) = \emptyset$ and, for $S \neq \emptyset$, $\alpha^{df}(S) = a$ such that for $v \in \mathsf{dom}(\tau)$, $a(v) = \{int\}$ if $\tau(v) = int$, $a(v) = \{bool\}$ if $\tau(v) = bool$ and $a(v) = \{\mu(\phi(v)).\kappa \mid \langle \phi, \mu \rangle \in S$ and $\phi(v) \in Loc\}$ if $\tau(v) \in Class\_ref$. In this last case, α^{df} collects all the classes of the objects bound to v in some concrete state.*

$$\mathsf{nop}^{df}_\tau(\phi) = \phi \qquad (\mathsf{get\_int}^i_\tau)^{df}(\phi) = \phi[\{int\}/res]$$
$$(\mathsf{get\_nil}^{cr}_\tau)^{df}(\phi) = \phi[\emptyset/res] \qquad (\mathsf{get\_bool}^b_\tau)^{df}(\phi) = \phi[\{bool\}/res]$$
$$(\mathsf{get\_var}^v_\tau)^{df}(\phi) = \phi[\phi(v)/res] \qquad (\mathsf{put\_var}^v_\tau)^{df}(\phi) = \phi[\phi(res)/v]|_{-res}$$
$$(\mathsf{get\_field}^f_\tau)^{df}(\phi) = \begin{cases} \emptyset & \text{if } \phi(res) = \emptyset \\ \phi[\downarrow(d(\tau(res)).\tau(f))/res] & \text{otherwise} \end{cases}$$
$$(\mathsf{put\_field}^f_{\tau,\tau'})^{df}(\phi_1)(\phi_2) = \begin{cases} \emptyset & \text{if } \phi_1(res) = \emptyset \\ \phi_2|_{-res} & \text{otherwise} \end{cases}$$
$$=^{df}_\tau(\phi_1)(\phi_2) = \phi_2[\{bool\}/res] \qquad \cup^{df}_\tau(\phi_1)(\phi_2) = \lambda v \in \mathsf{dom}(\tau).\phi_1(v) \cup \phi_2(v)$$
$$(\mathsf{scope}^{mr,v_1,\dots,v_n}_\tau)^{df}(\phi) = [\iota_1 \mapsto \phi(v_1),\dots,\iota_n \mapsto \phi(v_n), \mathtt{this} \mapsto \phi(res)]$$
$$\text{where } \langle \iota_1,\dots,\iota_n\rangle = p(mr).s \setminus \{n(mr), \mathtt{this}\}$$
$$(\mathsf{unscope}^{mr}_\tau)^{df}(\phi_1)(\phi_2) = \phi_1[\phi_2(n(mr))/res]$$
$$(\mathsf{restrict}^{vs}_\tau)^{df}(\phi) = \phi|_{-vs} \qquad (\mathsf{expand}^{v:t}_\tau)^{df}(\phi) = \phi[\mathsf{init}^{df}(t)/v]$$
$$(\mathsf{lookup}^{id,mr}_\tau)^{df}(\phi) = \begin{cases} \phi[S/res] \\ \quad \text{if } S = \{cr \in \phi(res) \mid d(cr).\nu(id) = mr\} \neq \emptyset \\ \emptyset \\ \quad \text{otherwise} \end{cases}$$
$$(\mathsf{new}^{cr}_\tau)^{df}(\phi) = \phi[\{cr\}/res] \qquad \mathsf{is\_true}^{df}_\tau(\phi) = \mathsf{is\_false}^{df}_\tau(\phi) = \phi\ ,$$

where $\mathsf{init}^{df}(int) = \{int\}$, $\mathsf{init}^{df}(bool) = \{bool\}$, $\mathsf{init}^{df}(cr) = \emptyset$ for $cr \in Class\_ref$.

Figure4. The instantiation over $State^{df}$ of the operations of Figure 2 and of $\cup$.

Proposition 4. *The optimal abstract counterparts of the powerset extension of the operations of Fig. 2 and of $\cup$ are those given in Fig. 4. All of them, except $\cup^{df}_\tau$, are strict on all their arguments. For instance, $(\mathsf{unscope}^{mr}_\tau)^{df}(\emptyset)(\phi_2) = (\mathsf{unscope}^{mr}_\tau)^{df}(\phi_1)(\emptyset) = (\mathsf{unscope}^{mr}_\tau)^{df}(\emptyset)(\emptyset) = \emptyset$. For $\cup^{df}_\tau$, we define $\cup^{df}_\tau(\emptyset)(\phi_2) = \phi_2$, $\cup^{df}_\tau(\phi_1)(\emptyset) = \phi_1$ and $\cup^{df}_\tau(\emptyset)(\emptyset) = \emptyset$.*

Proof (Sketch). As in the proof of Prop. 2, for the case of $\mathsf{get\_field}$ we must compute $\alpha^{df}(\mathsf{get\_field}^f_\tau(\gamma^{df}(\phi)))$ with $\phi \in State^{df}_\tau$. If $\phi = \emptyset$ then $\mathsf{get\_field}^f_\tau(\gamma^{df}(\phi)) = \mathsf{get\_field}^f_\tau(\emptyset) = \emptyset$ and its abstraction is still $\emptyset$ (Prop. 3). Otherwise, let $\tau' = \tau[d(\tau(res)).\tau(f)/res]$ and $v \in \mathsf{dom}(\tau')$. We have $A = \alpha^{df}(\mathsf{get\_field}^f_\tau(\gamma^{df}(\phi))) = \alpha^{df}(\mathsf{get\_field}^f_\tau(\{\langle\phi',\mu\rangle \in State_\tau \mid \phi \approx_\mu \phi'\})) = \alpha^{df}(\{\langle\phi'[(\mu(\phi'(res)).\phi)(f)/res],\mu\rangle \in State_{\tau'} \mid \langle\phi',\mu\rangle \in State_\tau,\ \phi'(res) \neq nil \text{ and } \phi \approx_\mu \phi'\})$. If $\phi(res) = \emptyset$, there is no $\langle\phi',\mu\rangle \in State_\tau$ such that $\phi \approx_\mu \phi'$ and $\phi'(res) \neq nil$. Then $A = \emptyset$. Otherwise $A(v)$ is $\{int\}$ if $\tau'(v) = int$, it is $\{bool\}$ if $\tau'(v) = bool$ and it is

$$\begin{cases} \left\{ \mu(l).\kappa \;\middle|\; \begin{array}{l} \langle\phi',\mu\rangle \in State_\tau,\ \phi'(res) \neq nil,\ \phi \approx_\mu \phi' \\ \text{and } (l = (\mu(\phi'(res)).\phi)(f)) \in Loc \end{array} \right\} & \begin{array}{l} \text{if } \tau'(v) \in Class\_ref \\ \text{and } v = res \end{array} \\ \left\{ \mu(\phi'(v)).\kappa \;\middle|\; \begin{array}{l} \langle\phi',\mu\rangle \in State_\tau,\ \phi'(res) \neq nil \\ \phi \approx_\mu \phi' \text{ and } \phi'(v) \in Loc \end{array} \right\} & \begin{array}{l} \text{if } \tau'(v) \in Class\_ref \\ \text{and } v \neq res. \end{array} \end{cases}$$

In the third case above, since fields are not approximated, it can be shown that $\mu(l).\kappa$ ranges exactly over $\downarrow(t)$, where $t = d(\tau(res)).\tau(f)$ is the declared type of the field f. In the fourth case above, it can be shown that the condition $\phi'(res) \neq nil$ can be deleted. Thus, we can rewrite those cases as $\downarrow(t)$ and $\alpha^{df}(\gamma^{df}(\phi))(v) = \phi(v)$, respectively, which leads to the operation in Figure 4.

Example 4. We execute over $State^{df}_{\tau}$ the same sequence of instructions of Ex. 2. The initial state is the abstraction (Prop. 3) of the initial concrete state.

Operation	State
	$[v_1 \mapsto \{\kappa_a\}]$ (initial state)
$(\mathsf{expand}^{v_2:\kappa_a}_{[v_1\mapsto\kappa_a]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset]$
$(\mathsf{new}^{\kappa_b}_{[v_1,v_2\mapsto\kappa_a]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset, res \mapsto \{\kappa_b\}]$
$(\mathsf{get\_field}^{n}_{[v_1,v_2\mapsto\kappa_a,res\mapsto\kappa_b]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset, res \mapsto \{\kappa_a, \kappa_b\}]$
$(\mathsf{put\_var}^{v_2}_{[v_1,v_2,res\mapsto\kappa_a]})^{df}$	$[v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \{\kappa_a, \kappa_b\}]$

The final abstract state says that v_1 can be bound to objects of class a and v_2 to objects of class a or b, which is a correct approximation of the concrete final state of Example 2. It is strictly more precise than that obtained in Example 3.

6 0-CFA (*ps*-analysis)

In [11], Palsberg and Schwartzbach defined a class analysis as a constraint problem. Every variable and every field is given a set of classes. These sets are then related by constraints which model the dataflow of the program. This means that the information about program variables is that used in the case of the dataflow analysis of Section 5, but this analysis deals with fields too, though all objects of the same class are identified. The analysis of a method is the same for every call point and hence this analysis is a *0-CFA analysis* [13].

That analysis leads to an abstract domain made of an abstract frame, identical to those used in Section 5, and of an abstract memory. The abstract memory is an abstract frame for the fields of the classes. We can assume without any loss of generality that fields in different classes have different names.

The difference with the original technique in [11] is that abstract interpretation does not *a priori* identify different occurrences of the same variable, while in [11] the same variable has assigned the same set of classes in every program point. This, together with the formally proved optimality of our abstract operations (Proposition 6), suggests that our analysis should be more precise than that in [11], without using any technique like multiple variables for different uses of the same variable or method splitting [12].

Definition 9. *We define the typing* $\overline{\tau} = \cup_{\kappa \in Class\_ref} d(\kappa).\tau$ *of the fields of all classes. It makes sense since fields have different names. For* $\tau \in Typing$ *such that* $\mathsf{dom}(\tau) \subseteq \mathsf{dom}(\overline{\tau})$ *and* $\phi \in Frame_{\tau}$, *we define* $\overline{\phi} \in Frame_{\overline{\tau}}$ *as* $\overline{\phi}(v) = \phi(v)$ *if* $v \in \mathsf{dom}(\tau)$, *and* $\overline{\phi}(v) = \mathsf{init}(\overline{\tau}(v))$ *otherwise (*init *has been defined in Fig. 2).*

Definition 10. *For* $\tau \in Typing$, *the abstract domain is* $State^{ps}_{\tau} = (Frame^{ps}_{\tau} \times Memory^{ps})$, *with* $Frame^{ps}_{\tau} = State^{df}_{\tau}$ *and* $Memory^{ps} = Frame^{df}_{\overline{\tau}}$ *(Defs. 5 and 6).*

$$
\begin{array}{rlrl}
\mathsf{nop}_\tau^{ps}(\langle\phi,\mu\rangle) &= \langle\phi,\mu\rangle & (\mathsf{get\_int}_\tau^{i})^{ps}(\langle\phi,\mu\rangle) &= \langle\phi[\{int\}/res],\mu\rangle\\
(\mathsf{get\_nil}_\tau^{cr})^{ps}(\langle\phi,\mu\rangle) &= \langle\phi[\emptyset/res],\mu\rangle & (\mathsf{get\_bool}_\tau^{b})^{ps}(\langle\phi,\mu\rangle) &= \langle\phi[\{bool\}/res],\mu\rangle\\
(\mathsf{get\_var}_\tau^{v})^{ps}(\langle\phi,\mu\rangle) &= \langle\phi[\phi(v)/res],\mu\rangle & (\mathsf{put\_var}_\tau^{v})^{ps}(\langle\phi,\mu\rangle) &= \langle\phi[\phi(res)/v]|_{-res},\mu\rangle
\end{array}
$$

$$
(\mathsf{get\_field}_\tau^{f})^{ps}(\langle\phi,\mu\rangle) = \begin{cases}\langle\emptyset,\mu\rangle & \text{if } \phi(res)=\emptyset\\ \langle\phi[\mu(f)/res],\mu\rangle & \text{otherwise}\end{cases}
$$

$$
(\mathsf{put\_field}_{\tau,\tau'}^{f})^{ps}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \begin{cases}\langle\emptyset,\mu_2\rangle\\ \quad\text{if } \phi_1(res)=\emptyset\\ \langle\phi_2|_{-res},\mu_2[\mu_2(f)\cup\phi_2(res)/f]\rangle\\ \quad\text{otherwise}\end{cases}
$$

$$
=_\tau^{ps}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \langle\phi_2[\{bool\}/res],\mu_2\rangle
$$

$$
(\mathsf{scope}_\tau^{mr,v_1,\dots,v_n})^{ps}(\langle\phi,\mu\rangle) = \langle[\iota_1\mapsto\phi(v_1),\dots,\iota_n\mapsto\phi(v_n),\mathtt{this}\mapsto\phi(res)],\mu\rangle
$$
$$
\text{where } \langle\iota_1,\dots,\iota_n\rangle = p(mr).s\setminus\{n(mr),\mathtt{this}\}
$$

$$
(\mathsf{unscope}_\tau^{mr})^{ps}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \langle\phi_1[\phi_2(n(mr))/res],\mu_2\rangle
$$

$$
(\mathsf{restrict}_\tau^{vs})^{ps}(\langle\phi,\mu\rangle) = \langle\phi|_{-vs},\mu\rangle \qquad (\mathsf{expand}_\tau^{v:t})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\mathsf{init}^{df}(t)/v],\mu\rangle
$$

$$
(\mathsf{lookup}_\tau^{id,mr})^{ps}(\langle\phi,\mu\rangle) = \begin{cases}\langle\phi[S/res],\mu\rangle\\ \quad\text{if } S=\{cr\in\phi(res)\mid d(cr).\nu(id)=mr\}\neq\emptyset\\ \langle\emptyset,\mu\rangle\\ \quad\text{otherwise}\end{cases}
$$

$$
(\mathsf{new}_\tau^{cr})^{ps}(\langle\phi,\mu\rangle) = \langle\phi[\{cr\}/res],\mu\rangle \qquad \mathsf{is\_true}_\tau^{ps}(\langle\phi,\mu\rangle) = \mathsf{is\_false}_\tau^{ps}(\langle\phi,\mu\rangle) = \langle\phi,\mu\rangle
$$

$$
\cup_\tau^{ps}(\langle\phi_1,\mu_1\rangle)(\langle\phi_2,\mu_2\rangle) = \langle\cup_\tau^{df}(\phi_1)(\phi_2),\cup_{\overline{\tau}}^{df}(\mu_1)(\mu_2)\rangle\ ,
$$

where init^{df} and $\cup^{df}$ are defined in Figure 4.

Figure5. The instantiation over $State^{ps}$ of the operations of Fig. 2 and of $\cup$.

Compare the following definition with Definition 8. The relation $\approx$ has been defined in Definition 7. An element $\langle\phi,\mu\rangle\in State_\tau^{ps}$ represents those concrete states whose frame is compatible with ϕ, i.e., their class variables are bound to objects of a class allowed by ϕ, and whose memory contains objects with an internal frame compatible with μ, i.e., their fields are bound to objects of a class allowed by μ. Since the frame of an object is for its instance variables only, we extend it to the whole set of fields before its comparison with μ.

Definition 11. *Given $\tau\in Typing$, we define the concretisation map γ^{ps} : $State_\tau^{ps}\mapsto\wp(State_\tau)$ as*

$$
\gamma^{ps}(\langle\phi,\mu\rangle) = \phi\cap\left\{\langle\phi',\mu'\rangle\in State_\tau\ \middle|\ \begin{array}{l}\textit{for every } \langle\kappa,\phi''\rangle\in\mathsf{codom}(\mu')\\ \textit{we have } \mu\approx_{\mu'}\overline{\phi''}\end{array}\right\}\ .
$$

The following abstraction map formalises the idea of the analysis. A set of concrete states S is abstracted through α^{df}, as in Section 5, but a second component provides more information about the fields. Namely, it is the abstraction through α^{df} of the states of the objects in memory.

Proposition 5. *For $\tau\in Typing$, the set $\gamma^{ps}(State_\tau^{ps})$ is a Moore family of $\wp(State_\tau)$. Hence $State_\tau^{ps}$ is an abstract domain whose induced abstraction map* *is* $\alpha^{ps} : \wp(State_\tau) \mapsto State^{ps}_\tau$ *such that* ($\mathsf{init}(\overline{\tau})$ *avoids empty abstract memories*)

$$\alpha^{ps}(S) = \langle \alpha^{df}(S), \alpha^{df}(\{\mathsf{init}(\overline{\tau})\} \cup \{\langle \overline{\phi'}, \mu\rangle \mid \langle \phi, \mu\rangle \in S \text{ and } \langle \kappa, \phi'\rangle \in \mathsf{codom}(\mu)\})\rangle.$$

Proposition 6. *The optimal abstract counterparts of the powerset extension of the operations of Fig. 2 and of* $\cup$ *are given in Fig. 5 for the case when the frame of their arguments is not* $\emptyset$. *Otherwise they yield* $\langle \emptyset, \mu\rangle$ *for an arbitrary* $\mu \in Memory^{ps}$, *except* $\cup^{ps}$ *which is such that* $\cup^{ps}_\tau(\langle \emptyset, \mu\rangle)(e) = \cup^{ps}_\tau(e)(\langle \emptyset, \mu\rangle) = e$.

In Figure 5, the frame component of the abstract states behaves like in Section 5, except in get_field. The memory component does not change if the memory of the concrete operations (Figure 2) does not change. The operations get_field and put_field allow a flow of information between abstract frames and abstract memories. Namely, get_field loads in the abstract frame the set of classes $\mu(f)$ of the objects stored in the field f. Conversely, put_field adds the classes contained in res, i.e., $\phi_2(res)$, to the set of classes already stored in the field, i.e., $\mu_2(f)$. In this way, we accumulate all classes stored in the field during the execution.

Example 5. We execute over $State^{ps}_\tau$ the same sequence of instructions of Ex. 2. The initial state is the abstraction (Prop. 5) of the initial concrete state.

Operation	State
	$\langle [v_1 \mapsto \{\kappa_a\}], [\mathtt{n} \mapsto \emptyset]\rangle$ (initial state)
$(\mathsf{expand}^{v_2:\kappa_a}_{[v_1 \mapsto \kappa_a]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset]\rangle$
$(\mathsf{new}^{\kappa_b}_{[v_1, v_2 \mapsto \kappa_a]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset, res \mapsto \{\kappa_b\}], [\mathtt{n} \mapsto \emptyset]\rangle$
$(\mathsf{get\_field}^{\mathtt{n}}_{[v_1, v_2 \mapsto \kappa_a, res \mapsto \kappa_b]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset, res \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset]\rangle$
$(\mathsf{put\_var}^{v_2}_{[v_1, v_2, res \mapsto \kappa_a]})^{ps}$	$\langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset]\rangle$

The final state says that v_1 can be bound to objects of class a, and that v_2 and the field n are bound to nil, the exact approximation of the final state of Ex. 2.

7 Comparison

We compare here the precision of the domains defined in the previous sections.

Proposition 7. *For* $\tau \in Typing$, *we have* $\gamma^{df}(State^{df}_\tau) \subseteq \gamma^{ps}(State^{ps}_\tau)$ *and the ps-analysis is more precise than the df-analysis.*

Proof. Since $\gamma^{df}(\emptyset) = \emptyset = \gamma^{ps}(\langle \emptyset, \mu\rangle) \in \gamma^{ps}(State^{ps}_\tau)$ for every $\mu \in Memory^{ps}$, we show that $\gamma^{df}(Frame^{df}_\tau) \subseteq \gamma^{ps}(State^{ps}_\tau)$. The idea is that every $\phi \in Frame^{df}_\tau$ is the frame of a pair $\langle \phi, \mu\rangle \in State^{ps}_\tau$ where μ does not introduce any restriction. Let $\mu(v) = \{int\}$ if $\overline{\tau}(v) = int$, $\mu(v) = \{bool\}$ if $\overline{\tau}(v) = bool$ and $\mu(v) = \downarrow(\overline{\tau}(v))$ if $\overline{\tau}(v) \in Class\_ref$ for $v \in \mathsf{dom}(\overline{\tau})$. Then $\langle \phi', \mu'\rangle \in \gamma^{df}(\phi)$ iff $\phi \approx_{\mu'} \phi'$ iff $\phi \approx_{\mu'} \phi'$ and for every $\langle \kappa, \phi''\rangle \in \mathsf{codom}(\mu')$ we have $\mu \approx_{\mu'} \overline{\phi''}$, iff $\langle \phi', \mu'\rangle \in \gamma^{ps}(\langle \phi, \mu\rangle)$, i.e., $\gamma^{df}(\phi) = \gamma^{ps}(\langle \phi, \mu\rangle)$. That inclusion entails the result about the precision of the analyses, since the abstract operations (Figs. 4 and 5) are optimal (§8 of [9]).

The *rta*- and the *df*-analysis are incomparable. In the final states of Exs. 3 and 4 the *df*- is more precise than the *rta*-analysis. But consider the initial states of those examples, i.e., the respective abstractions of the initial concrete state of Ex. 2. In Ex. 3 the possible classes for the field `n` are $\{\kappa_a\}$, while in Ex. 4 they are $\{\kappa_a, \kappa_b\}$, because that abstraction does not provide any information about the fields. Hence the *df*-analysis provides a coarser approximation of the set of classes for the field `n` than the *rta*-analysis.

Proving that the *ps*-analysis is more precise than the *rta*-analysis is more complex. The *ps*-analysis distinguishes between the classes stored in different variables, while the *rta*-analysis merges all classes in just one set. For instance, in the context of Ex. 1, the concrete states $\sigma_1 = \langle [v_1 \mapsto l, v_2 \mapsto nil], [l \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ and $\sigma_2 = \langle [v_1 \mapsto nil, v_2 \mapsto l], [l \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ are such that $\alpha^{rta}(\{\sigma_1\}) = \alpha^{rta}(\{\sigma_2\}) = \{\kappa_a\}$ while $\alpha^{ps}(\{\sigma_1\}) = \langle [v_1 \mapsto \{\kappa_a\}, v_2 \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$ and $\alpha^{ps}(\{\sigma_2\}) = \langle [v_1 \mapsto \emptyset, v_2 \mapsto \{\kappa_a\}], [\mathtt{n} \mapsto \emptyset] \rangle$. In this case, the *ps*-analysis distinguishes σ_1 and σ_2 which do contain different class information, while the *rta*-analysis does not. When trying to generalise this result into a formal proof, we are faced with the problem that the *rta*-analysis considers all objects in memory, while the *ps*-analysis only those reachable from the current frame or that of another object (Def. 11). For instance, in the context of Ex. 1, the concrete states $\sigma_3 = \langle [v \mapsto nil], [l \mapsto \langle \kappa_a, [\mathtt{n} \mapsto nil] \rangle] \rangle$ and $\sigma_4 = \langle [v \mapsto nil], [] \rangle$ are such that $\alpha^{ps}(\{\sigma_3\}) = \alpha^{ps}(\{\sigma_4\}) = \langle [v \mapsto \emptyset], [\mathtt{n} \mapsto \emptyset] \rangle$, while $\alpha^{rta}(\{\sigma_3\}) = \{\kappa_a\}$ and $\alpha^{rta}(\{\sigma_4\}) = \emptyset$. However, this ability to distinguish such states, which contain *the same* class information, is of no use for the class analysis. Indeed, objects in memory can affect an expression of the program only if they are reachable from some variable. A formal proof of this statement can be obtained, e.g., by quotienting [5] both domains w.r.t. class information.

8 Conclusions

Class analyses for object-oriented languages are many and varied. This paper shows that abstract interpretation leads to a formal framework for the development and comparison of such analyses. To demonstrate this, we have put three traditional techniques for class analysis inside that framework. This provides a systematic construction of the abstract operations of a particular analysis. Furthermore, it allows a formal comparison of the relative precision of the analyses. Due to space limitations, it has not been possible to show the details of how the set of operations can be used to give semantics to *e.g.*, Java or Java bytecode.

Several extensions to the present work are to be considered:

- There are of course more class analyses than the ones considered in this paper and the classification that we have initiated here should be carried further.
- Analyses can be combined so as to use the combination of two abstract domains in an analysis. Formally, this is characterised by the *reduced product*

of two abstractions that defines a semi-lattice structure on the set of abstractions. An interesting problem is to construct a concrete representation of the reduced product of, say, the *rta*- and the *df*-analysis.

- Since we have algorithms for the abstraction of a finite set of states (Props. 1, 3 and 5), and for the abstract operations (Props. 2, 4 and 6) and since our domains are finite (for a given $\tau \in Typing$), the framework provides a way of implementing the static class analyses. However, the practical aspects of such an implementation remain to be investigated.

References

1. O. Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In B. Le Charlier, editor, *Proc. of the 1st Int. Static Analysis Symp.*, volume 864 of *Lecture Notes in Computer Science*, pages 78–100. Springer-Verlag, 1994.
2. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
3. P. Bertelsen. Semantics of Java Byte Code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.
4. E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim, J. Grunska, and J. Zlatusla, editors, *23rd Int. Symp. on Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*. Springer, 1998.
5. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998.
6. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. In S. Brookes and M. Mislove, editors, *13th Conf. on Math. Found. of Programming Semantics*, volume 6 of *Electronic Notes on Theoretical Computer Science*, Pittsburgh, PA, USA, March 1997. Elsevier Science Publishers. Available at `http://www.elsevier.nl/locate/entcs/volume6.html`.
7. P. Cousot. Types as Abstract Interpretations. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press, 1997.
8. 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 Symp. on Principles of Programming Languages*, pages 238–252, 1977.
9. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *6th ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
10. A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and Effective Analysis of Statically Typed Object-Oriented Programs. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 292–305, New York, 1996. ACM Press.
11. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.
12. J. Plevyak and A. A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proc. of OOPSLA'94*, volume 29(10) of *ACM SIGPLAN Notices*, pages 324–340. ACM Press, October 1994.
13. O. Shivers. Control-Flow Analysis in Scheme. In *Proc. of the 1988 Conf. on Programming Languages Design and Implementation*, volume 23(7) of *ACM SIGPLAN Notices*, pages 164–174. ACM Press, July 1988.