Higher-Order Symb Comput (2006) 19:415–463
DOI 10.1007/s10990-006-0481-5

Deriving escape analysis by abstract interpretation

Patricia M. Hill · Fausto Spoto

Abstract Escape analysis of object-oriented languages approximates the set of objects which do not *escape* from a given context. If we take a method as context, the non-escaping objects can be allocated on its activation stack; if we take a thread, Java synchronisation locks on such objects are not needed. In this paper, we formalise a basic escape domain $\mathcal{E}$ as an abstract interpretation of concrete states, which we then refine into an abstract domain $\mathcal{ER}$ which is more concrete than $\mathcal{E}$ and, hence, leads to a more precise escape analysis than $\mathcal{E}$. We provide optimality results for both $\mathcal{E}$ and $\mathcal{ER}$, in the form of Galois insertions from the concrete to the abstract domains and of optimal abstract operations. The Galois insertion property is obtained by restricting the abstract domains to those elements which do not contain *garbage*, by using an abstract garbage collector. Our implementation of $\mathcal{ER}$ is hence an implementation of a formally correct escape analyser, able to detect the stack allocatable creation points of Java (bytecode) applications.

Keywords Abstract interpretation · Denotational semantics · Garbage collection

1 Introduction

Escape analysis identifies, at compile-time, some run-time data structures which do not *escape* from a given context, in the sense that they are not reachable anymore from that context. It has been studied for functional [5, 19, 36] as well as for object-oriented languages [1, 6, 7, 12, 21, 37–39, 41, 46, 49–51]. It allows one to stack allocate dynamically created data structures which would normally be heap allocated. This is possible if these data structures do not *escape* from the method which created them. Stack allocation reduces garbage collection overhead

P. M. Hill (✉)
University of Leeds, United Kingdom
e-mail: hill@comp.leeds.ac.uk

F. Spoto
Università di Verona, Italy
e-mail: fausto.spoto@univr.it

at run-time w.r.t. heap allocation, since stack allocated data structures are automatically deallocated when methods terminate. If, moreover, such data structures do not occur in a loop and their size is statically determined, they can be preallocated on the activation stack, which further improves the efficiency of the code. In the case of Java, which uses a mutual exclusion lock for each object in order to synchronise accesses from different threads of execution, escape analysis allows one also to remove unnecessary synchronisations, thereby making run-time accesses faster. By removing the space for the mutual exclusion lock associated with some of the objects, escape analysis can also help with space constraints. To this purpose, the analysis must prove that an object is accessed by at most one thread. This is possible if the object does not *escape* its creating thread.

Consider for instance our running example in Fig. 1. This program defines geometric figures `Square` and `Circle` that can be rotated. The class `Scan` has two methods, `scan` and `rotate`. The method `scan` calls `rotate` on each figure of a sequence `n` of figures passed as a parameter, as well as on a new `Square` and a new `Circle`. Each `new` statement is a creation point and has been decorated with a label, such as π_1 or π_2. We often abuse notation and call the label a creation point itself. Hence, we can say that the creation points π_2 and π_3 can be stack allocated since they create objects which are not reachable once the method `scan` terminates. On the contrary, the creation point π_4 cannot be allocated in the activation stack of the method `rotate`, since the `Angle` it creates is actually stored inside the field `rotation` of the `Square` object created at π_2, which is still reachable when `rotate` terminates. However, if we created `Circles` rather than `Squares` at π_2, and if we assumed that the `scan` method is passed a list of `Circles` as parameter, then the creation point π_4 could be stack allocated, since the virtual call `f.rot(a)` would always lead to the method `rot` inside `Circle`, which does not store its parameter in any field. The creation point π_1 cannot be stack allocated since it creates objects that are stored in the `rotation` field, and hence are still reachable when the method completes. Note that we assume here that an object escapes from a method if it is still *reachable* when the method terminates. Others [38, 41, 50] require that that object is actually *used* after the method terminates. Our assumption is more conservative and hence leads to less precise analyses. However, it lets us analyse libraries, whose calling contexts are not known at analysis time, so that it is undetermined whether an object is actually used after a library method terminates or not.

1.1 Contributions of our work

This paper presents two escape analyses for Java programs. The goal of both analyses is to detect objects that do not escape (i.e., are unreachable from outside) a certain scope. This information can later be used to stack-allocate captured (i.e., non-escaping) objects.

Both analyses use the object allocation site model: all objects allocated at a given program point (possibly in a loop) are modelled by the same creation point. The first analysis, based on the abstract domain $\mathcal{E}$, expresses the information we need for our stack allocation. Namely, for each program point, it provides an over-approximation of the set of creation points that escape because they are transitively reachable from a set of escapability roots (i.e., variables including parameters, static fields, method result). The domain $\mathcal{E}$ does not keep track of other information such as the creation points pointed to by each individual variable or field.

Although $\mathcal{E}$ is the property neede for stack allocation, a static analysis based on $\mathcal{E}$ is not sufficiently precise as it does not relate the creation points with the variables and fields that point to them. We therefore consider a refinement $\mathcal{ER}$ of $\mathcal{E}$ that preserves this information and also includes $\mathcal{E}$ so that $\mathcal{ER}$ contains just the minimum information needed for stack allocation.

```
class Angle {
 int degree;     int acute() { return this.degree < 90; }
}
class Figure {
 Figure next;
 void def() {}     void rot(Angle a) {}     void draw() {}
}
class Square extends Figure {
 int side, x, y;     Angle rotation;
 void def() {
  this.side = 1; this.x = this.y = 0;
  this.rotation = new Angle();     {π1}
  this.rotation.degree = 0;
 }
 void rot(Angle a) { this.rotation = a; }
 void draw() { ... use this.rotation here ... }
}
class Circle extends Figure {
 int radius, x, y;
 void def() {
  this.radius = 1; this.x = this.y = 0;     {w0}
 }
 void draw() { ... }
}
class Scan {
 void scan(Figure n) {
  Figure f = new Square();     {π2}
  f.next = f;     {w1}
  f.def(); rotate(f);
  f = new Circle();     {π3}
  f.def();     {w2}
  f.next = n;
  while (f != null) { rotate(f); f = f.next; }
 }
 void rotate(Figure f) {
  Angle a = new Angle();     {π4}
  f.rot(a); a.degree = 0;
  while (a.degree < 360) { a.degree++; f.draw(); }
 }
}
```

Fig. 1 Running example

Both analyses are developed in the abstract interpretation framework [14, 15], and we present proofs that the associated transfer functions are optimal with respect to the abstractions that are used by each analysis i.e., they make the best possible use of the abstract information expressed by the abstract domains.

To increase the precision of the two analyses and to get a Galois insertion, rather than a Galois connection, both analyses use local variable scoping and type information. Hence,

the abstract domains contain no spurious element. We achieve this goal through *abstract garbage collectors* which remove some elements from the abstract domains whenever they reflect unreachable (and hence, for our analysis, irrelevant) portions of the run-time heap, as also [12] does, although [12] does not relate this to the Galois insertion property. Namely, the abstract domains are exactly the set of fixpoints of their respective abstract garbage collectors and, hence, do not contain spurious elements.

The contribution of this paper is a clean construction of an escape analysis through abstract interpretation thus obtaining formal and detailed proofs of correctness as well as optimality. Optimality states that the abstract domains are related to the concrete domain by a Galois *insertion*, rather than just a *connection* and in the use of optimal abstract operations. Precision and efficiency of the analysis are not the main issues here, although we are pleased to see that our implementation scales to relatively large applications and compares well with some already existing and more precise escape analyses (Section 6).

1.2 The basic domain $\mathcal{E}$

Our work starts by defining a basic abstract domain $\mathcal{E}$ for escape analysis. Its definition is guided by the observation that a creation point π occurring in a method m can be stack allocated if the objects it creates are not reachable at the end of m from a set of variables E which includes m's return value, the fields of the objects bound to its formal parameters at call-time (including the implicit `this` parameter) and any exceptions thrown by m. Note that we consider the fields of the objects bound to the formal parameters at call-time since they are aliases of the actual arguments, and hence still reachable when the method returns. For a language, such as Java, which allows static fields, E also includes the static fields. Variables with integer type are not included in E since no object can be reached from an integer. Moreover, local variables are also not included in E since local variables accessible inside a method m will disappear once m terminates. The basic abstract domain $\mathcal{E}$ is hence defined as the collection of all sets of creation points. Each method is decorated with an element of $\mathcal{E}$, which contains precisely the creation points of the objects reachable from the variables in E at the end of the method.

Example 1. Consider for instance the method `scan` in Fig. 1. We assume that *null* is passed to `scan` as a parameter, and that its `this` object has been created at an external creation point $\overline{\pi}$. We have $\mathcal{E} = \wp(\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\})$ and $E = \{\mathtt{n}\}$ (the implicit parameter `this` has type `Scan` and hence no fields, so that there is no need to consider it in E). Then `scan` is decorated with $\varnothing$ since no object can be reached at the end of `scan` from the variables in $E = \{\mathtt{n}\}$. Consequently, the creation points π_2 and π_3 can be stack allocated since they do not belong to $\varnothing$. Note that, if `n` had been modified inside the method `scan` then we would have used $E = \{\mathtt{n}'\}$, where $\mathtt{n}'$ is a *shadow copy* of `n` which holds its *initial* value (we will see this technique in Example 52).

We still have to specify how this decoration is computed for each method. We use abstract interpretation to propagate an input set of creation points through the statements of each method, until its end is reached. This is accomplished by defining a *transfer function* for every statement of the program which, in terms of abstract interpretation, is called an *abstract operation* (see Section 4 and Fig. 9). The element of $\mathcal{E}$ resulting at the end of each method is then *restricted* to the appropriate set E for that method through an abstract operation called restrict. By applying the theory of abstract interpretation, we know that this restriction is a conservative approximation of the actual decoration we need at the end of each method.

Fig. 2 The propagation of $\{\overline{\pi}\}$ through the method `scan` of the program in Fig. 1

```
void scan(Figure n) {
{π̄}
 Figure f = new Square();   {π2}
{π̄, π2}
 f.next = f;   {w1}
{π̄, π2}
 f.def();
{π̄, π1, π2}
 rotate(f);
{π̄, π1, π2, π4}
 f = new Circle();   {π3}
{π̄, π1, π2, π3, π4}
 f.def();
{π̄, π1, π2, π3, π4}
 f.next = n;
{π̄, π1, π2, π3, π4}
 while(f != null) {
{π̄, π1, π2, π3, π4}
  rotate(f);
{π̄, π1, π2, π3, π4}
  f = f.next;
{π̄, π1, π2, π3, π4}
 }
}
```

Example 2. Consider again Example 1 and the method `scan`. In Fig. 2 we propagate the set $\{\overline{\pi}\}$ through `scan`'s statements by following the translation of high-level statements into the *bytecodes* as specified in Fig. 9. The restriction of the set $\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$ to $E = \{\mathtt{n}\}$ is $\{\pi_1, \pi_2, \pi_3, \pi_4\}$ (objects of class `Scan` created at $\overline{\pi}$ are incompatible with the type of `n`), which is a very imprecise approximation of the desired result i.e., $\varnothing$.

The problem here is that although the abstract domain $\mathcal{E}$ expresses the kind of decoration we need for stack allocation, $\mathcal{E}$ has very poor computational properties. In terms of abstract interpretation, it induces very imprecise abstract operations and, just as in the case of the basic domain $\mathcal{G}$ for *groundness analysis* of logic programs [32], it needs refining [22, 42].

Nevertheless, it must be observed that the abstract domain $\mathcal{E}$ already contains some non-trivial information. For instance, since $\overline{\pi}$ is a creation point for objects of class `Scan` and π_2 is a creation point for objects of class `Square`, then the first `f.def()` virtual call occurring in the method `scan` can only lead to the method `def` inside `Square`. Hence we say that *our escape information contains information on the run-time late-binding mechanism*, which can be exploited to improve the precision of the analysis by refining the call-graph. This is what actually happens in Example 2. Note also that the local scope may temporarily introduce new variables so that at the end of the scope, any creation points that can *only* be reached from these variables can be safely removed from the approximation. In Fig. 3, the approximation computed at p_2 is $\{\overline{\pi}, \pi_s, \pi_1\}$, where $\overline{\pi}$ is the creation point of `this`, but the approximation computed at p_3 is $\{\overline{\pi}\}$, which is smaller. For this reason, we say that *our escape information uses the static type information* to improve the precision of the analysis. That is, the possible

Fig. 3 A program fragment where the set of variables in scope grows and shrinks

```
{only this of type Scan is in scope here}
{
  Figure f = new Square();        {π_s}
  {p_1}
  f.def();      {this creates an object at π_1 (Figure 1)}
  {p_2}
}
{p_3}
```

approximations from $\mathcal{E}$ in a given program point, are constrained by the (finite) set of variables and their types that are in scope.

We formalise the fact that the approximation in $\mathcal{E}$ can shrink, by means of an *abstract garbage collector* (Definition 25) i.e., a garbage collector that works over sets of creation points instead of concrete objects. When a variable's scope is closed, the abstract garbage collector removes from the approximation of the next statement all creation points which can *only* be reached *from that variable*. The name of abstract garbage collector is justified by the fact that this conservatively maintains in the approximation the creation points of the objects which *might* be reachable in the concrete state, thus modeling in the abstract domain a behaviour similar to that of a concrete garbage collector. It must be noted, however, that our abstract garbage collector only considers reachability from the variables in scope in the current method, while a concrete garbage collector would consider reachability from all variables in the current activation stack.

The abstract garbage collector is of no use in the propagation of $\bar{\pi}$ shown in Fig. 2 since, for instance, after the creation point π_3, it is not possible to conclude that the object o created at π_2, and hence also those created at π_1 and π_4 and stored in o's `rotation` field, are not reachable anymore. This is because $\mathcal{E}$ does not distinguish between the objects reachable from variables `f` and `n`. Before π_3, the object o created at π_2 can be reached from `f` only, but π_3 overwrites `f` so it cannot be reached anymore. Because $\mathcal{E}$ does not make a distinction between objects reachable from `f` and `n`, it cannot infer this, because it considers that o could be reachable from `n`. The only safe choice is to be conservative and assume that it cannot be garbage collected.

1.3 The refinement $\mathcal{ER}$

The abstract domain $\mathcal{E}$ represents the information we need for stack allocation, but it does not include any other related information that may improve the precision of the abstract operations, such as explicit information about the creation points of the objects bound to *a given* variable or field. However, the ability to reason on a per variable basis is essential for the precision of a static analysis of imperative languages, where assignment to a given variable or field is the basic computational mechanism. So we *refine* $\mathcal{E}$ into a new abstract domain $\mathcal{ER}$ which splits the sets of creation points in $\mathcal{E}$ into subsets, one for each variable or field. We show that $\mathcal{ER}$ strictly contains $\mathcal{E}$, justifying the name of *refinement*.

We perform a static analysis based on $\mathcal{ER}$ exactly as for $\mathcal{E}$ but using the abstract operations for the domain $\mathcal{ER}$ given in Section 5 (see Fig. 10).

Example 3. Consider again the method `scan` in Fig. 1. We start the analysis from the element $[\texttt{this} \mapsto \{\overline{\pi}\}]\star[]$ of $\mathcal{ER}$ which expresses the fact that the variable `this` is initially bound

```
void scan(Figure n) {
[this ↦ {π̄}] ⋆[]
 Figure f = new Square();   {π2}
[f ↦ {π2}, this ↦ {π̄}] ⋆[]
 f.next = f;
[f ↦ {π2}, this ↦ {π̄}] ⋆[next ↦ {π2}]
 f.def();
[f ↦ {π2}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1}]
 rotate(f);
[f ↦ {π2}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1, π4}]
 f = new Circle();   {π3}
[f ↦ {π3}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1, π4}]
 f.def();
[f ↦ {π3}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1, π4}]
 f.next = n;
[f ↦ {π3}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1, π4}]
 while(f != null) {
[f ↦ {π2, π3}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1, π4}]
   rotate(f);
[f ↦ {π2, π3}, this ↦ {π̄}] ⋆[next ↦ {π2}, rotation ↦ {π1, π4}]
   f = f.next;
 }
[this ↦ {π̄}] ⋆[]
}
```

Fig. 4 The propagation of [this ↦ {$\overline{\pi}$}] ⋆[] through the method scan of the program in Fig. 1

to an object created at the external creation point $\overline{\pi}$ and all other variables and fields are initially bound to *null* (if they have class type) or to an integer (otherwise). The operator $\star$ is a pair-separator; its component [this ↦ {$\overline{\pi}$}] is the approximation for the variables in scope and its component [] is the approximation for the fields. The information is then propagated, as shown in Fig. 4. Then, just as for the domain $\mathcal{E}$, at the end of the method the result [this ↦ {$\overline{\pi}$}] ⋆[] is restricted to $E = \{\mathtt{n}\}$ and we get [] ⋆[], which leads to $\varnothing$ which is a much more precise approximation than the set $\{\pi_1, \pi_2, \pi_3, \pi_4\}$ obtained in Example 2 with $\mathcal{E}$.

Note that at the end of the method (when f and n go out of scope), the approximation of the fields next and rotation are reset to $\varnothing$. The justification for this is that, at this point, it is no longer possible to reach the Square object created at π_2 whose field rotation contained objects created at π_1 or π_4, nor is it possible to reach the Circle object created at π_3 whose next field might have contained something created at π_2. This is an example of the application of our abstract garbage collector for $\mathcal{ER}$ (Definition 44).

The domain $\mathcal{ER}$ can hence be seen as the specification of a new escape analysis, which includes $\mathcal{E}$ as its foundational kernel. Example 3 shows that the abstract domain $\mathcal{ER}$ is actually more precise than $\mathcal{E}$. Our implementation of $\mathcal{ER}$ (Section 6) shows that it can actually be used to obtain non-trivial escape analysis information for Java bytecode.

1.4 Structure of the paper

After a brief summary of our notation and terminology in Section 2, we pass in Section 3 to recall the framework of [45] on which the analysis is based. Then, in Section 4, we formalise our basic domain $\mathcal{E}$ and provide suitable abstract operations for its analysis. We show that the analysis induced by $\mathcal{E}$ is very imprecise. Hence, in Section 5 we refine the domain $\mathcal{E}$ into the more precise domain $\mathcal{ER}$ for escape analysis. In Section 6, we discuss our prototype implementation and experimental results. Section 7 discusses related work. Section 8 concludes the main part of the paper. Proofs not inlined in this paper are available in [29].

Preliminary, partial versions of this paper appeared in [26] and [27]. The current paper is a seamless fusion of these papers, with the proofs of the theoretical results and with a description and evaluation of the implementation of the escape analysis over the domain $\mathcal{ER}$.

2 Preliminaries

A total (partial) function f is denoted by $\mapsto$ ($\rightarrow$). The *domain* (*range*) of f is $\mathsf{dom}(f)$ ($\mathsf{rng}(f)$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $\mathsf{dom}(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \mathsf{dom}(f)$ (to $\mathsf{dom}(f) \setminus s$). If f and g are functions, we denote by fg the composition of f and g, such that $fg(x) = f(g(x))$. If $f(x) = x$ then x is a *fixpoint* of f. The set of fixpoints of f is denoted by $\mathsf{fp}(f)$.

A *pair* of elements is written $a \star b$. A definition of a pair S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$. The cardinality of a set S is denoted by $\#S$. The *disjoint union* of two sets S, T is denoted by $S + T$. To simplify expressions, particulary when the set is used as a subscript, we sometimes write a singleton set $\{x\}$ as x. If S is a set and $\leq$ is a partial relation over S, we say that S is a *partial ordering* if it is reflexive ($s \leq s$ for every $s \in S$), transitive ($s_1 \leq s_2$ and $s_2 \leq s_3$ entail $s_1 \leq s_2$ for every $s_1, s_2, s_3 \in S$) and anti-symmetric ($s_1 \leq s_2$ and $s_2 \leq s_1$ entail $s_1 = s_2$ for every $s_1, s_2 \in S$). If S is a set and $\leq$ a partial ordering on S, then the pair $S \star \leq$ is a *poset*.

A *complete lattice* is a poset $C \star \leq$ where *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist. Let $C \star \leq$ and $A \star \preceq$ be posets and $f : C \mapsto A$. We say that f is *monotonic* if $c_1 \leq c_2$ entails $f(c_1) \preceq f(c_2)$. It is *(co-)additive* if it preserves lub's (glb's). Let $f : A \mapsto A$. The map f is *reductive* (respectively, *extensive*) if $f(a) \preceq a$ (respectively, $a \preceq f(a)$) for any $a \in A$. It is *idempotent* if $f(f(a)) = f(a)$ for any $a \in A$. It is a *lower closure operator* (*lco*) if it is *monotonic*, *reductive* and *idempotent*.

We recall now the basics of abstract interpretation [14, 15]. Let $C \star \leq$ and $A \star \preceq$ 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 $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. It is a *Galois insertion* when $\alpha\gamma$ is the identity map i.e., when the abstract domain does not contain *useless* elements. If C and A are complete lattices and α is strict and additive, then α is the abstraction map of a Galois connection. If, moreover, α is onto or γ is one-to-one, then α is the abstraction map of a Galois insertion. In a Galois connection, γ can be defined in terms of α as $\gamma(a) = \cup\{c \mid \alpha(c) \preceq a\}$, where $\cup$ is the least upper bound operation over the concrete domain C. Hence, it is enough to provide α to define a Galois connection. An abstract operator $\hat{f} : A^n \mapsto A$ is *correct* w.r.t. $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$. For each operator f, there exists an *optimal* (most precise) correct abstract operator $\hat{f}$ defined as $\hat{f} = \alpha f \gamma$. This means that $\hat{f}$ does the best it can with the information expressed by the abstract domain.

The composition of correct operators is correct. The composition of optimal operators is not necessarily optimal. The *semantics* of a program is the fixpoint of a map $f : C \mapsto C$, where C is the *computational domain*. Its *collecting version* [14, 15] 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 suboperations, their powerset extensions *and* $\cup$ (which merges the semantics of the branches of a conditional) induce the extension of f.

3 The framework of analysis

The framework presented here is for a simple typed object-oriented language where the concrete states and operations are based on [45]. It allows us to derive a compositional, denotational semantics, which can be seen as an analyser, from a specification of a domain of abstract states and operations which work over them (hence called *state transformers*). Then problems such as scoping, recursion and name clash can be ignored, since these are already solved by the semantics. Moreover, this framework relates the precision of the analysis to that of its abstract domain so that traditional techniques for comparing the precision of abstract domains can be applied [13–15].

The definition of a denotational semantics, in the style of [52], by using the state transformers of this section can be found in [45]. Here we only want to make clear some points:

- We allow expressions to have side-effects, such as method call expressions, which is not the case in [52]. As a consequence, the evaluation of an expression from an initial state yields both a final state *and* the value of the expression. We use a special variable *res* of the final state to hold this value;
- The evaluation from an initial state σ_1 of a binary operation such as $e_1 + e_2$, where e_1 and e_2 are expressions, first evaluates e_1 from σ_1, yielding an intermediate state σ_2, and then evaluates e_2 from σ_2, yielding a state σ_3. The value v_1 of *res* in σ_2 is that of e_1, and the value v_2 of *res* in σ_3 is that of e_2. We then modify σ_3 by storing in *res* the sum $v_1 + v_2$. This yields the final state. Note that the single variable *res* is enough for this purpose. The complexity of this mechanism w.r.t. a more standard approach [52] is, again, a consequence of the use of expressions with side-effects;
- Our denotational semantics deals with method calls through *interpretations*: an interpretation is the input/output behaviour of a method, and is used as its denotation whenever that method is called. As a nice consequence, our states contain only a single frame, rather than an activation stack of frames. This is standard in denotational semantics and has been used for years in logic programming [8].
- The computation of the semantics of a program starts from a bottom interpretation which maps every input state to an undefined final state and then updates this interpretation with the denotations of the methods body. This process is iterated until a fixpoint is reached as is done for logic programs [8]. The same technique can be applied to compute the abstract semantics of a program, but the computation is performed over the abstract domain. It is also possible to generate constraints which relate the abstract approximations at different program points, and then solve such constraints with a fixpoint engine. The latter is the technique that we use in Section 6.

3.1 Programs and creation points

We recall here the semantical framework of [45].

Definition 4 (Type Environment). Each program in the language has a finite set of *identifiers* *Id* such that `out`, `this` $\in$ *Id* and a finite set of *classes* $\mathcal{K}$ ordered by a *subclass relation* $\leq$ such that $\mathcal{K}\star\leq$ is a poset. Let *Type* $= \{int\} \uplus \mathcal{K}$ and $\leq$ be extended to *Type* by defining $int \leq int$. Let *Vars* $\subseteq$ *Id* be a set of *variables* such that $\{\texttt{out}, \texttt{this}\} \subseteq Vars$. A *type environment* for a program is any element of the set

$$TypEnv = \{\tau : Vars \to Type \mid \text{if } \texttt{this} \in \mathsf{dom}(\tau) \text{ then } \tau(\texttt{this}) \in \mathcal{K}\}.$$

In the following, τ will implicitly stand for a type environment.

A class contains local variables (*fields*) and functions (*methods*). A method has a set of input/output variables called *parameters*, including `out`, which holds the result of the method, and `this`, which is the object over which the method has been called (the *receiver* of the call). Methods returning `void` are represented as methods returning an *int* of constant value 0, implicitly ignored by the caller of the method.

Example 5. Consider the example program given in Fig. 1. Here *Id* includes, in addition to the identifiers `out` and `this`, user-defined identifiers such as `rotation`, `def`, `x`, `y`, `rot`. The set of classes is

$$\mathcal{K} = \{\texttt{Angle}, \texttt{Figure}, \texttt{Square}, \texttt{Circle}, \texttt{Scan}\}$$

where the ordering $\leq$ is defined so that `Square` $\leq$ `Figure` and `Circle` $\leq$ `Figure`. Variables for this program include `x`, `y`, `f`, `n` and `this`. Variable `out` is used to hold the return value of the methods, so that the `return` e statement can be seen as syntactic sugar for `out` $= e$ (with no following statements). At points w_0 and w_1, the type environments are

$$\tau_{w_0} = [\texttt{out} \mapsto int, \texttt{this} \mapsto \texttt{Circle}]$$

$$\tau_{w_1} = [\texttt{f} \mapsto \texttt{Figure}, \texttt{n} \mapsto \texttt{Figure}, \texttt{out} \mapsto int, \texttt{this} \mapsto \texttt{Scan}].$$

Fields is a set of maps which bind each class to the type environment of its fields. The variable `this` cannot be a field. *Methods* is a set of maps which bind each class to a map from identifiers to methods. *Pars* is a set of maps which bind each method to the type environment of its parameters (its signature).

Definition 6 (Field, Method, Parameter). Let $\mathcal{M}$ be a finite set of *methods*. We define

$$Fields = \{F : \mathcal{K} \mapsto TypEnv \mid \texttt{this} \notin \mathsf{dom}(F(\kappa)) \text{ for every } \kappa \in \mathcal{K}\}$$

$$Methods = \mathcal{K} \mapsto (Id \to \mathcal{M})$$

$$Pars = \{P : \mathcal{M} \mapsto TypEnv \mid \{\texttt{out}, \texttt{this}\} \subseteq \mathsf{dom}(P(\nu)) \text{ for } \nu \in \mathcal{M}\}.$$

The *static information* of a program is used by the static analyser.

Definition 7 (Static Information). The *static information* of a program consists of a poset $\mathcal{K}\star\leq$, a set of methods $\mathcal{M}$ and maps $F \in Fields$, $M \in Methods$ and $P \in Pars$.

Fields in different classes but with the same name can be disambiguated by using their *fully qualified name* such as in the Java Virtual Machine [33]. For instance, we write `Circle.x` for the field `x` of the class `Circle`.

$$\mathcal{K} = \left\{ \begin{array}{l} \texttt{Angle}, \\ \texttt{Figure}, \\ \texttt{Square}, \\ \texttt{Circle}, \\ \texttt{Scan} \end{array} \right\} \quad \mathcal{M} = \left\{ \begin{array}{l} \texttt{Angle.acute}, \\ \texttt{Figure.def}, \texttt{Figure.rot}, \texttt{Figure.draw}, \\ \texttt{Square.def}, \texttt{Square.rot}, \texttt{Square.draw}, \\ \texttt{Circle.def}, \texttt{Circle.draw}, \\ \texttt{Scan.scan}, \texttt{Scan.rotate} \end{array} \right\}$$

$\texttt{Square} \leq \texttt{Figure}, \quad \texttt{Circle} \leq \texttt{Figure}$ and reflexive cases

$$F(\texttt{Angle}) = [\texttt{degree} \mapsto \mathit{int}] \qquad F(\texttt{Scan}) = []$$

$$F(\texttt{Figure}) = [\texttt{next} \mapsto \texttt{Figure}]$$

$$F(\texttt{Square}) = \left[\begin{array}{l} \texttt{side} \mapsto \mathit{int}, \texttt{Square.x} \mapsto \mathit{int}, \texttt{Square.y} \mapsto \mathit{int}, \\ \texttt{rotation} \mapsto \texttt{Angle}, \texttt{next} \mapsto \texttt{Figure} \end{array} \right]$$

$$F(\texttt{Circle}) = \left[\begin{array}{l} \texttt{radius} \mapsto \mathit{int}, \texttt{Circle.x} \mapsto \mathit{int}, \\ \texttt{Circle.y} \mapsto \mathit{int}, \texttt{next} \mapsto \texttt{Figure} \end{array} \right]$$

$$M(\texttt{Angle}) = [\texttt{acute} \mapsto \texttt{Angle.acute}]$$

$$M(\texttt{Figure}) = \left[\begin{array}{l} \texttt{def} \mapsto \texttt{Figure.def}, \texttt{rot} \mapsto \texttt{Figure.rot}, \\ \texttt{draw} \mapsto \texttt{Figure.draw} \end{array} \right]$$

$$M(\texttt{Square}) = \left[\begin{array}{l} \texttt{def} \mapsto \texttt{Square.def}, \texttt{rot} \mapsto \texttt{Square.rot}, \\ \texttt{draw} \mapsto \texttt{Square.draw} \end{array} \right]$$

$$M(\texttt{Circle}) = \left[\begin{array}{l} \texttt{def} \mapsto \texttt{Circle.def}, \texttt{rot} \mapsto \texttt{Figure.rot}, \\ \texttt{draw} \mapsto \texttt{Circle.draw} \end{array} \right]$$

$$M(\texttt{Scan}) = [\texttt{scan} \mapsto \texttt{Scan.scan}, \texttt{rotate} \mapsto \texttt{Scan.rotate}]$$

$$P(\texttt{Angle.acute}) = [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{Angle}]$$

$$P(\texttt{Figure.rot}) = [\texttt{a} \mapsto \texttt{Angle}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{Figure}]$$

$$P(\texttt{Scan.rotate}) = [\texttt{f} \mapsto \texttt{Figure}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{Scan}]$$

$$P(\texttt{Figure.def}) = [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{Figure}]$$

(the other cases of P are as above)

Fig. 5 The static information of the program in Fig. 1

Example 8. The static information of the program in Fig. 1 is shown in Fig. 5. Note that the result of the method `Angle.acute` in Fig. 1 becomes the type of `out` in $P(\texttt{Angle.acute})$ in Fig. 5.

The only points in the program where new objects can be created are the `new` statements. We require that each of these statements is identified by a unique label called its *creation point*.

Definition 9 (Creation Point). Let Π be a finite set of labels called *creation points*. A map $k : \Pi \mapsto \mathcal{K}$ relates every creation point $\pi \in \Pi$ with the class $k(\pi)$ of the objects it creates.

Example 10. Consider again the program in Fig. 1. In that program $\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$ is the set of creation points, where we assume that $\overline{\pi}$ decorates an external creation point for `Scan` (not shown in the figure). Then

$$k = [\overline{\pi} \mapsto \texttt{Scan}, \pi_1 \mapsto \texttt{Angle}, \pi_2 \mapsto \texttt{Square}, \pi_3 \mapsto \texttt{Circle}, \pi_4 \mapsto \texttt{Angle}].$$

3.2 Concrete states

To represent the concrete state of a computation at a particular program point we need to refer to the concrete values that may be assigned to the variables. Apart from the integers and *null*, these values need to include *locations* which are the addresses of the memory cells used at that point. Then the concrete state of the computation consists of a map that assigns type consistent values to variables (*frame*) and a map from locations to objects (*memory*) where an *object* is characterised by its creation point and the frame of its fields. Hence the notion of object that we use here is more concrete than that in [45], which relates a *class* rather than a *creation point* to each object. A memory can be *updated* by assigning new (type consistent) values to the variables in its frames.

Definition 11 (Location, Frame, Object, Memory). Let *Loc* be an infinite set of *locations* and $Value = \mathbb{Z} + Loc + \{null\}$. We define *frames*, *objects* and *memories* as

$$Frame_\tau = \left\{ \phi \in \mathsf{dom}(\tau) \mapsto Value \;\middle|\; \begin{array}{l} \text{for every } v \in \mathsf{dom}(\tau) \\ \tau(v) = int \Rightarrow \phi(v) \in \mathbb{Z} \\ \tau(v) \in \mathcal{K} \Rightarrow \phi(v) \in \{null\} \cup Loc \end{array} \right\}$$

$$Obj = \{\pi \star \phi \mid \pi \in \Pi,\ \phi \in Frame_{F(k(\pi))}\}$$

$$Memory = \{\mu \in Loc \to Obj \mid \mathsf{dom}(\mu) \text{ is finite}\}.$$

Let $\mu_1, \mu_2 \in Memory$ and $L \subseteq \mathsf{dom}(\mu_1)$. We say that μ_2 is an *L-update* of μ_1, written $\mu_1 =_L \mu_2$, if $L \subseteq \mathsf{dom}(\mu_2)$ and for every $l \in L$ we have $\mu_1(l).\pi = \mu_2(l).\pi$.

The initial value for a variable of a given type is used when we add a variable in scope. It is defined as $\Im(int) = 0$, $\Im(\kappa) = null$ for $\kappa \in \mathcal{K}$. This function is extended to type environments (Definition 4) as $\Im(\tau)(v) = \Im(\tau(v))$ for every $v \in \mathsf{dom}(\tau)$.

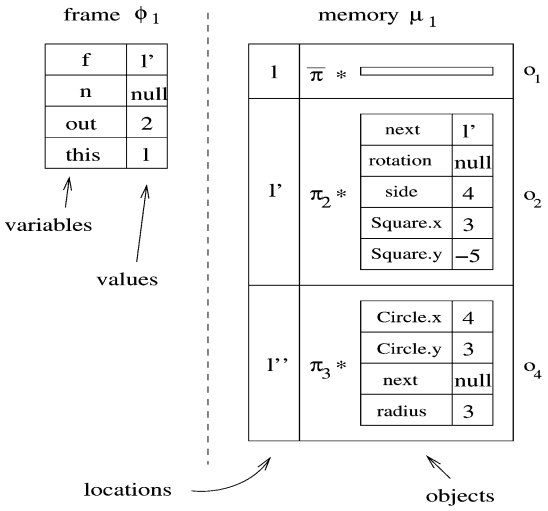
Example 12. Consider again the program in Fig. 1 and its static information in Fig. 5. The type environment τ_{w_1}, a frame ϕ_1, memories μ_1, μ_2, μ_3, objects o_1, o_2, o_3, o_4 and a state σ_1 for this program at program point w_1 with locations $l, l', l'' \in Loc$ are given in Fig. 6. Let also

$$\phi_2 = [\mathtt{f} \mapsto 2, \mathtt{n} \mapsto l', \mathtt{out} \mapsto -2, \mathtt{this} \mapsto l].$$

Then $Frame_{\tau_{w_1}}$ contains ϕ_1 but not ϕ_2 because `f` is bound to 2 in ϕ_2 (while it has class `Figure` in τ_{w_1}).

The object o_1 created at $\overline{\pi}$ has class $k(\overline{\pi}) = \mathtt{Scan}$ since $F(\mathtt{Scan}) = []$. Objects created at π_2 have class `Square` so that these could be o_2 and o_3. Similarly, since $k(\pi_3) = \mathtt{Circle}$, an object created at π_3 is o_4. With these objects, *Memory* contains the maps μ_1, μ_2, μ_3. With these definitions of μ_1, μ_2 and μ_3, the memory μ_2 is neither an l-update nor an l'-update of μ_1 since $\mu_1(l).\pi = \overline{\pi}$ whereas $\mu_2(l).\pi = \pi_2$ and also $\mu_1(l').\pi = \pi_2$ whereas $\mu_2(l').\pi = \overline{\pi}$. However, as $\mu_3(l).\pi = \pi_2$ and $\mu_3(l').\pi = \pi_2$, we have $\mu_1 =_{l'} \mu_3$ and $\mu_2 =_l \mu_3$. Also, letting $\mu_4 = [l \mapsto o_1, l' \mapsto o_3, l'' \mapsto o_4]$, then we have $\mu_1 =_{\{l,l',l''\}} \mu_4$.

Type correctness and conservative garbage collection guarantee that there are no dangling pointers and that variables may only be bound to locations which contain objects allowed by



$$
\begin{aligned}
k &= [\overline{\pi} \mapsto \texttt{Scan}, \pi_1 \mapsto \texttt{Angle}, \pi_2 \mapsto \texttt{Square}, \pi_3 \mapsto \texttt{Circle}, \pi_4 \mapsto \texttt{Angle}],\\
\tau_{w_1} &= [\texttt{f} \mapsto \texttt{Figure}, \texttt{n} \mapsto \texttt{Figure}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{Scan}],\\
\phi_1 &= [\texttt{f} \mapsto l', \texttt{n} \mapsto \mathit{null}, \texttt{out} \mapsto 2, \texttt{this} \mapsto l],\\
o_1 &= \overline{\pi} \star [],\\
o_2 &= \pi_2 \star \begin{bmatrix} \texttt{next} \mapsto l', \texttt{rotation} \mapsto \mathit{null}, \\ \texttt{side} \mapsto 4, \texttt{Square.x} \mapsto 3, \texttt{Square.y} \mapsto -5 \end{bmatrix},\\
o_3 &= \pi_2 \star \begin{bmatrix} \texttt{next} \mapsto \mathit{null}, \texttt{rotation} \mapsto l, \\ \texttt{side} \mapsto 4, \texttt{Square.x} \mapsto 3, \texttt{Square.y} \mapsto -5 \end{bmatrix},\\
o_4 &= \pi_3 \star [\texttt{Circle.x} \mapsto 4, \texttt{Circle.y} \mapsto 3, \texttt{next} \mapsto \mathit{null}, \texttt{radius} \mapsto 3],\\
\mu_1 &= [l \mapsto o_1, l' \mapsto o_2, l'' \mapsto o_4],\\
\mu_2 &= [l \mapsto o_2, l' \mapsto o_1],\\
\mu_3 &= [l \mapsto o_2, l' \mapsto o_3],\\
\sigma_1 &= \phi_1 \star \mu_1.
\end{aligned}
$$

Fig. 6 The creation point map k and, for program point w_1, the type environment, a frame, objects and memories for the program in Fig. 1

the type environment. This is a sensible constraint for the memory allocated by strongly-typed languages such as Java [3].

Definition 13 (Weak Correctness). Let $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is *weakly τ-correct* w.r.t. μ if for every $v \in \mathsf{dom}(\phi)$ such that $\phi(v) \in \mathit{Loc}$ we have $\phi(v) \in \mathsf{dom}(\mu)$ and $k((\mu\phi(v)).\pi) \leq \tau(v)$.

We strengthen the correctness notion of Definition 13 by requiring that it also holds for the fields of the objects in memory.

Definition 14 (τ-Correctness). Let $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is *τ-correct* w.r.t. μ and write $\phi \star \mu : \tau$, if

1. ϕ is weakly τ-correct w.r.t. μ and,
2. for every $o \in \mathsf{rng}(\mu)$, $o.\phi$ is weakly $F(k(o.\pi))$-correct w.r.t. μ.

Example 15. Let τ_{w_1}, ϕ_1, μ_1, μ_2 and μ_3 be as in Fig. 6.

– $\phi_1 \star \mu_1 : \tau_{w_1}$. Condition 1 of Definition 14 holds because

$$\{v \in \mathsf{dom}(\phi_1) \mid \phi_1(v) \in \mathit{Loc}\} = \{\mathtt{this}, \mathtt{f}\},$$

$$\{\phi_1(\mathtt{this}), \phi_1(\mathtt{f})\} = \{l, l'\} \subseteq \mathsf{dom}(\mu_1),$$

$$k(\mu_1(l).\pi) = k(o_1.\pi) = k(\overline{\pi}) = \mathtt{Scan} = \tau_{w_1}(\mathtt{this}),$$

$$k(\mu_1(l').\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{Square} \leq \mathtt{Figure} = \tau_{w_1}(\mathtt{f}).$$

Condition 2 of Definition 14 holds because

$$\mathsf{rng}(\mu_1) = \{o_1, o_2, o_4\},$$

$$\mathsf{rng}(o_1.\phi) = \mathsf{rng}(o_4.\phi) \cap \mathit{Loc} = \varnothing,$$

$$\mathsf{rng}(o_2.\phi) \cap \mathit{Loc} = \{l'\} \subseteq \mathsf{dom}(\mu_1),$$

$$k(\mu_1(l').\pi) = k(o_2.\pi) = \mathtt{Square} \leq \mathtt{Figure} = F(o_2.\pi)(\mathtt{next}).$$

– $\phi_1 \star \mu_2 : \tau_{w_1}$ does not hold, since condition 1 of Definition 14 does not hold. Namely, $\tau_{w_1}(\mathtt{this}) = \mathtt{Scan}$, $k((\mu_2\phi_1(\mathtt{this})).\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{Square}$ and $\mathtt{Square} \not\leq \mathtt{Scan}$.
– $\phi_1 \star \mu_3 : \tau_{w_1}$ does not hold, since condition 2 of Definition 14 does not hold. Namely, $o_3 \in \mathsf{rng}(\mu_3)$ and $o_3.\phi$ is not $F(k(o_3.\pi))$-correct w.r.t. μ_3, since we have that $o_3.\phi(\mathtt{rotation}) = l$, $\mathtt{Square} \not\leq \mathtt{Angle}$ but $k(\mu_3(l).\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{Square}$ and moreover $F(k(o_3.\pi))(\mathtt{rotation}) = F(\mathtt{Square})(\mathtt{rotation}) = \mathtt{Angle}$.

Definition 16 defines the state of the computation as a pair consisting of a frame and a memory. The variable `this` in the domain of the frame must be bound to an object. In particular, it cannot be *null*. This condition could be relaxed in Definition 16. This would lead to simplifications in the following sections (such as in Definition 25). However, our condition is consistent with the specification of the Java programming language [3]. Note, however, that there is no such hypothesis about the local variable number 0 of the Java Virtual Machine, which stores the `this` object [33].

Definition 16 (State). If τ is a type environment associated with a program point, the set of possible *states* of a computation at that point is any subset of

$$\Sigma_\tau = \left\{ \phi \star \mu \;\middle|\; \begin{array}{l} \phi \in \mathit{Frame}_\tau,\ \mu \in \mathit{Memory},\ \phi \star \mu : \tau, \\ \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ then } \phi(\mathtt{this}) \neq \mathit{null} \end{array} \right\}.$$

Example 17. Let τ_{w_1}, ϕ_1, μ_1, μ_2 and μ_3 be as in Fig. 6. Then, in Example 15, we have shown that $\phi_1 \star \mu_1 : \tau_{w_1}$ holds and that $\phi_1 \star \mu_2 : \tau_{w_1}$ and $\phi_1 \star \mu_3 : \tau_{w_1}$ do not hold. Thus, at program point w_1, we have $\phi_1 \star \mu_1 \in \Sigma_{\tau_{w_1}}$, $\phi_1 \star \mu_2 \notin \Sigma_{\tau_{w_1}}$ and $\phi_1 \star \mu_3 \notin \Sigma_{\tau_{w_1}}$.

The frame of an object o in memory is itself a state for the instance variables of o.

Proposition 18. *Let $\phi \star \mu \in \Sigma_\tau$ and $o \in \mathsf{rng}(\mu)$. Then $(o.\phi) \star \mu \in \Sigma_{F(k(o.\pi))}$.*

Proof: Since $\phi \star \mu \in \Sigma_\tau$, from Definition 16 we have $\phi \star \mu : \tau$. From Definition 14 we know that $o.\phi$ is weakly $F(k(o.\pi))$-correct w.r.t. μ so that $(o.\phi) \star \mu : F(k(o.\pi))$. Since $\texttt{this} \notin \mathsf{dom}(F(k(o.\pi)))$ (Definition 6) we conclude that $(o.\phi) \star \mu \in \Sigma_{F(k(o.\pi))}$.

□

3.3 The operations over the concrete states

Figures 7 and 8 show the signatures and the definitions, respectively, of a set of operations over the concrete states for a type environment τ. The variable *res* holds intermediate results, as we said at the beginning of this section. We briefly introduce these operations.

Operation	Constraint ($\texttt{this} \in \mathsf{dom}(\tau)$ always)
$\mathsf{nop}_\tau : \Sigma_\tau \mapsto \Sigma_\tau$	
$\mathsf{get\_int}^i_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto int]}$	$res \notin \mathsf{dom}(\tau),\ i \in \mathbb{Z}$
$\mathsf{get\_null}^\kappa_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto \kappa]}$	$res \notin \mathsf{dom}(\tau),\ \kappa \in \mathcal{K}$
$\mathsf{get\_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto \tau(v)]}$	$res \notin \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau)$
$\mathsf{get\_field}^f_\tau : \Sigma_\tau \rightarrow \Sigma_{\tau[res \mapsto i(f)]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$, $i = F\tau(res),\ f \in \mathsf{dom}(i)$
$\mathsf{put\_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau\mid_{-res}}$	$res \in \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau)$, $v \neq res,\ \tau(res) \leq \tau(v)$
$\mathsf{put\_field}^f_{\tau,\tau'} : \Sigma_\tau \mapsto \Sigma_{\tau'} \rightarrow \Sigma_{\tau\mid_{-res}}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$ $f \in \mathsf{dom}(F\tau(res))$ $\tau' = \tau[res \mapsto t]$ with $t \leq (F\tau(res))(f)$
$=_\tau, +_\tau : \Sigma_\tau \mapsto \Sigma_\tau \mapsto \Sigma_\tau$	$res \in \mathsf{dom}(\tau),\ \tau(res) = int$
$\mathsf{is\_null}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto int]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$
$\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau : \Sigma_\tau \mapsto \Sigma_{P(\nu)\mid_{-\texttt{out}}}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$, $\{v_1,\ldots,v_n\} \subseteq \mathsf{dom}(\tau),\ \nu \in \mathcal{M}$ $\mathsf{dom}(P(\nu)) \setminus \{\texttt{out},\texttt{this}\} = \{\iota_1,\ldots,\iota_n\}$ (alphabetically ordered) $\tau(res) \leq P(\nu)(\texttt{this})$ $\tau(v_i) \leq P(\nu)(\iota_i)$ for $i = 1,\ldots,n$
$\mathsf{return}^\nu_\tau : \Sigma_\tau \mapsto \Sigma_{p\mid_{\texttt{out}}} \rightarrow \Sigma_{\tau[res \mapsto p(\texttt{out})]}$	$res \in \mathsf{dom}(\tau),\ \nu \in \mathcal{M},\ p = P(\nu)$
$\mathsf{restrict}^{vs}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau\mid_{-vs}}$	$vs \subseteq \mathsf{dom}(\tau)$
$\mathsf{expand}^{v:t}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[v \mapsto t]}$	$v \in \mathit{Vars},\ v \notin \mathsf{dom}(\tau),\ t \in \mathit{Type}$
$\mathsf{new}^\pi_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res \mapsto k(\pi)]}$	$res \notin \mathsf{dom}(\tau),\ \pi \in \Pi$
$\mathsf{lookup}^{m,\nu}_\tau : \Sigma_\tau \rightarrow \Sigma_{\tau[res \mapsto P(\nu)(\texttt{this})]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$, $m \in \mathsf{dom}(M\tau(res)),\ \nu \in \mathcal{M}$ for every suitable m, σ and τ, there is at most one ν such that $\mathsf{lookup}^{m,\nu}_\tau(\sigma)$ is defined
$\mathsf{is\_true}_\tau : \Sigma_\tau \rightarrow \Sigma_{\tau\mid_{-res}}$ $\mathsf{is\_false}_\tau : \Sigma_\tau \rightarrow \Sigma_{\tau\mid_{-res}}$	$res \in \mathsf{dom}(\tau),\ \tau(res) = int$, $\mathsf{dom}(\mathsf{is\_true}_\tau) \cap \mathsf{dom}(\mathsf{is\_false}_\tau) = \varnothing$ $\mathsf{dom}(\mathsf{is\_true}_\tau) \cup \mathsf{dom}(\mathsf{is\_false}_\tau) = \Sigma_\tau$

Fig. 7 The signature of the operations over the states

- The nop operation does nothing.
- A get operation loads into *res* a constant, the value of another variable or the value of the field of an object. In the last case (get_field), that object is assumed to be stored in *res before* the get operation. Then $(\mu\phi'(res))$ is the object whose field f must be read, $(\mu\phi'(res)).\phi$ are its fields and $(\mu\phi'(res)).\phi(f)$ is the value of the field named f.
- A put operation stores in v the value of *res* or of a field of an object pointed to by *res*. Note that, in the second case, put_field is a binary operation since the evaluation of $e_1.f = e_2$ from an initial state σ_1 works by first evaluating e_1 from σ_1, yielding an intermediate state σ_2, and then evaluating e_2 from σ_2, yielding a state σ_3. The final state is then $\mathsf{put\_field}(\sigma_2)(\sigma_3)$ [45], where the variable *res* of σ_2 holds the value of e_1 and the variable *res* of σ_3 holds the value of e_2. The object whose field is modified must still exist in the memory of σ_3. This is expressed by the update relation (Definition 11). As there is no result, *res* is removed. Providing two states i.e., two frames and two heaps for put_field and, more generally, for binary operations, may look like an overkill and it might be expected that a single state and a single frame would be enough. However, our decision to have two states has been dictated by the intended use of this semantics i.e., abstract interpretation. By *only* using operations over states, we have exactly one concrete domain, which can be abstracted into just one abstract domain. Hybrid operations, working on states and frames, would only complicate the abstraction.
- For every binary operation such as $=$ and $+$ over values, there is an operation on states. Note that (in the case of $=$) Booleans are implemented by means of integers (every non-negative integer means true). We have already explained why we use two states for binary operations.
- The operation is_null checks that *res* points to *null*.
- The operation call is used before, and the operation return is used after, a call to a method v. While call^v creates a new state in which v can execute, the operation return^v restores the state σ which was current before the call to v, and stores in *res* the result of the call. As said in (the beginning of) Section 3, the denotation of the method is taken from an *interpretation*, in a denotational fashion [8]. Hence the execution from an initial state σ_1 of a method call denoted, in the current interpretation, by $d : \Sigma \to \Sigma$, yields the final state $\mathsf{return}(\sigma_1)(d(\mathsf{call}(\sigma_1)))$. Note that return is a binary operation whose first argument is the state of the caller at call-time and whose second argument is the state of the callee at return-time. Its definition in Fig. 8 restores the state of the caller but stores in *res* the return value of the callee. By using a binary operation we can define our semantics in terms of states rather than in terms of activation stacks. This is a useful simplification when passing to abstraction, since states must be abstracted rather than stacks. Note that the update relation (Definition 11) requires that the variables of the caller have not been changed during the execution of the method (although the fields of the objects bound to those variables may be changed).
- The operation expand (restrict) adds (removes) variables.
- The operation new^π creates a new object o of creation point π. A pointer to o is put in *res*. Its fields are initialised to default values.
- The operation $\mathsf{lookup}^{m,v}$ checks if, by calling the method identified by m of the object o pointed to by *res*, the method v is run. This depends on the class $k(o.\pi)$ of $o = \mu\phi(res)$.
- The operation is_true (is_false) checks if *res* contains true (false).

Example 19. Consider again the example in Fig. 1. Let $\tau = \tau_{w_1}, \phi_1, \mu_1$ and $\sigma_1 = \phi_1 \star \mu_1$ be as in Fig. 6 so that, as indicated in Example 17, state σ_1 could be the current state at program

$$
\begin{aligned}
\mathsf{nop}_\tau(\phi \star \mu) &= \phi \star \mu \\
\mathsf{get\_int}^i_\tau(\phi \star \mu) &= \phi[res \mapsto i] \star \mu \\
\mathsf{get\_null}^\kappa_\tau(\phi \star \mu) &= \phi[res \mapsto null] \star \mu \\
\mathsf{get\_var}^v_\tau(\phi \star \mu) &= \phi[res \mapsto \phi(v)] \star \mu \\
\mathsf{restrict}^{vs}_\tau(\phi \star \mu) &= \phi|_{-vs} \star \mu \\
\mathsf{expand}^{v:t}_\tau(\phi \star \mu) &= \phi[v \mapsto \Im(t)] \star \mu \\
\mathsf{put\_var}^v_\tau(\phi \star \mu) &= \phi[v \mapsto \phi(res)]|_{-res} \star \mu \\
\mathsf{get\_field}^f_\tau(\phi' \star \mu) &= \begin{cases} \phi'[res \mapsto ((\mu\phi'(res)).\phi)(f)] \star \mu & \text{if } \phi'(res) \neq null \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathsf{put\_field}^f_{\tau,\tau'}(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= \begin{cases} \phi_2|_{-res} \star \mu_2[l \mapsto \mu_2(l).\pi \star \mu_2(l).\phi[f \mapsto \phi_2(res)]] \\ \quad \text{if } l = \phi_1(res), l \neq null \text{ and } \mu_1 =_l \mu_2 \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
=_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= \begin{cases} \phi_2[res \mapsto 1] \star \mu_2 & \text{if } \phi_1(res) = \phi_2(res) \\ \phi_2[res \mapsto -1] \star \mu_2 & \text{if } \phi_1(res) \neq \phi_2(res) \end{cases} \\
+_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= \phi_2[res \mapsto \phi_1(res) + \phi_2(res)] \star \mu_2 \\
\mathsf{is\_null}_\tau(\phi \star \mu) &= \begin{cases} \phi[res \mapsto 1] \star \mu & \text{if } \phi(res) = null \\ \phi[res \mapsto -1] \star \mu & \text{otherwise} \end{cases} \\
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(\phi \star \mu) &= [\iota_1 \mapsto \phi(v_1), \ldots, \iota_n \mapsto \phi(v_n), \mathtt{this} \mapsto \phi(res)] \star \mu \\
\text{where } \{\iota_1, \ldots, \iota_n\} &= P(\nu) \setminus \{\mathtt{out}, \mathtt{this}\} \text{ (alphabetically ordered)} \\
\mathsf{return}^\nu_\tau(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) &= \begin{cases} \phi_1[res \mapsto \phi_2(\mathtt{out})] \star \mu_2 \\ \quad \text{if } L = \mathsf{rng}(\phi_1)|_{-res} \cap Loc \text{ and } \mu_1 =_L \mu_2 \\ \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
\mathsf{new}^\pi_\tau(\phi \star \mu) &= \phi[res \mapsto l] \star \mu[l \mapsto \pi \star \Im(F(k(\pi)))],\ l \in Loc \setminus \mathsf{dom}(\mu) \\
\mathsf{lookup}^{m,\nu}_\tau(\phi \star \mu) &= \begin{cases} \phi \star \mu \\ \quad \text{if } \phi(res) \neq null \text{ and } M(k((\mu\phi(res)).\pi))(m) = \nu \\ \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
\mathsf{is\_true}_\tau(\phi \star \mu) &= \begin{cases} \phi|_{-res} \star \mu & \text{if } \phi(res) \geq 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathsf{is\_false}_\tau(\phi \star \mu) &= \begin{cases} \phi|_{-res} \star \mu & \text{if } \phi(res) < 0 \\ \text{undefined} & \text{otherwise.} \end{cases}
\end{aligned}
$$

Fig. 8 The operations over concrete states

point w_1. The computation continues as follows [45].

$$\sigma_2 = \mathsf{get\_var}^{\mathtt{f}}_\tau(\sigma_1) \qquad \text{read } \mathtt{f}.$$

Let o_1, o_2, o_3, o_4 be as in Fig. 6. Then $\sigma_2 = \phi_1\,[res \mapsto \phi_1(f)] \star \mu_1 = \phi_1\left[res \mapsto l'\right] \star \mu_1 = [\mathtt{f} \mapsto l', \mathtt{n} \mapsto null, \mathtt{out} \mapsto 2, res \mapsto l', \mathtt{this} \mapsto l] \star \mu_1$. The lookup operations determine which is the target of the first virtual call `f.def()` in Fig. 1. As a result only one of the

following blocks of code is run depending on which lookup check is defined.

$$
\begin{array}{ll}
\sigma_3' = \mathsf{lookup}^{\texttt{def,Figure.def}}_{\tau[res\mapsto \texttt{Figure}]}(\sigma_2) & \text{select } \texttt{Figure.def} \\
\sigma_4' = \mathsf{call}^{\texttt{Figure.def}}_{\tau[res\mapsto P(\texttt{Figure.def})(\texttt{this})]}(\sigma_3') & \text{initialise the } \texttt{Figure} \\
\sigma_5' = \text{the final state of } \texttt{Figure.def} \text{ from } \sigma_4' & \\
\sigma_6' = \mathsf{return}^{\texttt{Figure.def}}_{\tau[res\mapsto P(\texttt{Figure.def})(\texttt{this})]}(\sigma_3')(\sigma_5') & \text{return to the caller} \\
 & \\
\sigma_3'' = \mathsf{lookup}^{\texttt{def,Square.def}}_{\tau[res\mapsto \texttt{Figure}]}(\sigma_2) & \text{select } \texttt{Square.def} \\
\sigma_4'' = \mathsf{call}^{\texttt{Square.def}}_{\tau[res\mapsto P(\texttt{Square.def})(\texttt{this})]}(\sigma_3'') & \text{initialise the } \texttt{Square} \\
\sigma_5'' = \text{the final state of } \texttt{Square.def} \text{ from } \sigma_4'' & \\
\sigma_6'' = \mathsf{return}^{\texttt{Square.def}}_{\tau[res\mapsto P(\texttt{Square.def})(\texttt{this})]}(\sigma_3'')(\sigma_5'') & \text{return to the caller} \\
 & \\
\sigma_3''' = \mathsf{lookup}^{\texttt{def,Circle.def}}_{\tau[res\mapsto \texttt{Figure}]}(\sigma_2) & \text{select } \texttt{Circle.def} \\
\sigma_4''' = \mathsf{call}^{\texttt{Circle.def}}_{\tau[res\mapsto P(\texttt{Circle.def})(\texttt{this})]}(\sigma_3''') & \text{initialise the } \texttt{Circle} \\
\sigma_5''' = \text{the final state of } \texttt{Circle.def} \text{ from } \sigma_4''' & \\
\sigma_6''' = \mathsf{return}^{\texttt{Circle.def}}_{\tau[res\mapsto P(\texttt{Circle.def})(\texttt{this})]}(\sigma_3''')(\sigma_5''') & \text{return to the caller.}
\end{array}
$$

The states σ_5', σ_5'' and σ_5''' are computed from the current intepretation for the methods. For each lookup operation, we have $(\sigma_2.\phi)(res) = l' \neq null$ and $(\sigma_2.\mu)(l') = o_2$; then the method of o_2 identified by `def` is called. Now $o_2.\pi = \pi_2$ and $k(\pi_2) = \texttt{Square}$. Moreover $M(\texttt{Square})(\texttt{def}) = \texttt{Square.def}$ (Fig. 5). So the only *defined* lookup operation is that for `Square.def`. This means that `Square.def` is called and $\sigma_3'' = \sigma_2$, while σ_3' and σ_3''' are undefined.

We obtain $\sigma_4'' = [\texttt{this} \mapsto l'] \star \mu_1$. Note that the object o_2 is now the `this` object of this instantiation of the method `Square.def`. To compute σ_5'', we should execute the operations which implement `Square.def`, starting from the state σ_4''. For simplicity, we report only the final state of this execution which is

$$\sigma_5'' = [\texttt{out} \mapsto 0] \star \underbrace{[l \mapsto o_1, l' \mapsto o_5, l'' \mapsto o_4]}_{\mu_5},$$

where

$$o_5 = \pi_2 \star \begin{bmatrix} \texttt{next} \mapsto l', \texttt{side} \mapsto 1, \texttt{Square.x} \mapsto 0, \\ \texttt{Square.y} \mapsto 0, \texttt{rotation} \mapsto o_6 \end{bmatrix},$$

$$o_6 = \pi_1 \star [\texttt{degree} \mapsto 0].$$

The return operation returns the control to the caller method. This means that the frame will be that of the caller, but the return value of the callee is copied into the *res* variable of the caller. Namely,

$$\sigma_6'' = \begin{bmatrix} \texttt{f} \mapsto l', \texttt{n} \mapsto null, \\ \texttt{out} \mapsto 2, res \mapsto 0, \texttt{this} \mapsto l \end{bmatrix} \star \mu_5.$$

3.4 The collecting semantics

The operations of Fig. 8 can be used to define the transition function from states to states, or *denotation*, of a piece of code c, as shown in Example 19. By use of call and return,

there is a denotation for each method called in c; thus, by adding call and return, we can plug the method's denotation in the calling points inside c (as shown in Section 3.3 and in Example 19). A function I binding each method m in a program P to its denotation $I(\mathtt{m})$ is called an *interpretation* of P. Given an interpretation I, we are hence able to define the denotation $T_P(I)(\mathtt{m})$ of the body of a method m, so that we are able to transform I into a new interpretation $T_P(I)$. This leads to the definition of the *denotational semantics* of P as the minimal (i.e., less defined) interpretation which is a fixpoint of T_P. This way of defining the concrete semantics in a denotational way through interpretations, is useful for a subsequent abstraction [15]. The technique, which has been extensively used in the logic programming tradition [8], has been adapted in [45] for object-oriented imperative programs by adding the mechanism for dynamic dispatch through the lookup operation in Fig. 8. Note that the fixpoint of T_P is not finitely computable in general, but it does exist as a consequence of Tarski's theorem and it is the limit of the ascending chain of interpretations I_0, $T_P(I_0)$, $T_P(T_P(I_0))$, ... , where, for every method m, the denotation $I_0(\mathtt{m})$ is always undefined [47].

The concrete semantics described above denotes each method with a map on states i.e., a function from Σ to Σ. However, abstract interpretation is interested in *properties* of states; so that each property of interest, is identified with the set of all the states satisfying that property. This leads to the definition of a *collecting* semantics [14, 15] i.e., a concrete semantics working over the powerset $\wp(\Sigma)$. The operations of this collecting semantics are the powerset extension of the operations in Fig. 8. For instance, $\mathsf{get\_int}^i_\tau$ is extended into

$$\mathsf{get\_int}^i_\tau(S) = \{\mathsf{get\_int}^i_\tau(\sigma) \mid \sigma \in S\}$$

for every $S \in \wp(\Sigma_\tau)$. Note that dealing with powersets means that the semantics becomes non-deterministic. For instance, in Example 19 more than one target of the `f.def()` virtual call could be selected at the same time and more than one of the blocks of code could be executed. Hence we need a $\cup$ operation over sets of states which merges different threads of execution at the end of a virtual call (or, for similar motivations, at the end of a conditional). The notion of denotation now becomes a map over $\wp(\Sigma_\tau)$. Interpretations and the transformer on interpretations are defined exactly as above. We will assume the result, proved in [45], that every abstraction of $\wp(\Sigma_\tau)$, $\cup$ and of the powerset extension of the operations in Fig. 8 induces an abstraction of the concrete collecting semantics. This is an application to object-oriented imperative programs of the *fixpoint transfer* Proposition 27 in [15]. Two such abstractions will be described in Sections 4 and 5.

4 The basic domain $\mathcal{E}$

We define here a basic abstract domain $\mathcal{E}$ as a property of the concrete states of Definition 16. Its definition is guided by our goal to *over*approximate, for every program point p, the set of creation points of objects reachable at p from some variable or field in scope. Thus an element of the abstract domain $\mathcal{E}$ which decorates a program point p is simply a set of creation points of objects that may be reached at p. The choice of an *over*approximation follows from the typical use of the information provided by an escape analysis. For instance, an object can be stack allocated if it does not escape the method which creates it i.e., if it does not belong to a superset of the objects reachable at its end. Moreover, our goal is to stack allocate specific

creation points. Hence, we are not interested in the identity of the objects but in their creation points.

Although, at the end of this section, we will see that $\mathcal{E}$ induces rather imprecise abstract operations, its definition is important since $\mathcal{E}$ comprises exactly the information needed to implement our escape analysis. Even though its abstract operations lose precision, we still need $\mathcal{E}$ as a basis for comparison and as a minimum requirement for new, improved domains for escape analysis. Namely, in Section 5 we will define a more precise abstract domain $\mathcal{ER}$ for escape analysis, and we will prove (Proposition 56) that it strictly contains $\mathcal{E}$. This situation is similar to that of the abstract domain $\mathcal{G}$ for groundness analysis of logic programs [43] which, although imprecise, expresses the property looked for by the analysis, and is the basis of all the other abstract domains for groundness analysis, derived as *refinements* of $\mathcal{G}$ [42]. The definition of more precise abstract domains as refinements of simpler ones is actually standard methodology in abstract interpretation nowadays [22]. Another example is strictness analysis of functional programs, where a first simple domain is subsequently enriched to express more precise information [31]. A similar idea has also been applied to model-checking, through a sequence of refinements of a simple abstract domain [17]. A *refinement*, in this context, is just an operation that transforms a simpler domain into a richer one i.e., one containing more abstract elements. There are many standard refinements operations. One of this is *reduced product*, which allows one to compose two abstract domains in order to express the composition of the properties expressed by the two domains, and *disjunctive completion*, which enriches an abstract domain with the ability to express disjunctive information about the properties expressed by the domain [34]. Another example is the *linear refinement* of a domain w.r.t. another, which expresses the dependencies of the abstract properties expressed by the two domains [23]. In Section 5 we use a refinement which is significant for imperative programs, where assignments to program variables are the pervasive operation. Hence, a variable-based approximation often yields improved precision w.r.t. a global approximation of the state, such as expressed by $\mathcal{E}$. This same refinement is used, for instance, when passing from rapid type analysis to a variable-based *class analysis* of object-oriented imperative programs in [45].

We show an example now that clarifies the idea of *reachability* for objects at a program point.

Example 20. Consider the program in Fig. 1 and the type environment τ_{w_1} for program point w_1 given in Fig. 6. We show that no objects created at π_4 will be reachable at program point w_1. The type environment at w_1, which is τ_{w_1}, shows that we cannot reach any object from `out`, since `out` can only contain integers. The variable `this` has class `Scan` which has no fields. Since in π_4 we create objects of class `Angle`, they cannot be reached from `this`. The variables `f` and `n` have class `Figure` whose only field has type `Figure` itself. Reasoning as for `this`, we could falsely conclude that no object created at π_4 can be reached from `f` or `n`. This conclusion is false since, as `Square` $\leq$ `Figure`, the call `rotate(f)` could result in a call in the class `Square` to the method `f.rot(a)` which stores `a`, created at π_4, in the field `rotation`; and `rotation` is still accessible to other methods such as `f.draw()`.

The reasoning in Example 20 leads to the notion of *reachability* in Definition 21 where we use the actual fields of the objects instead of those of the declared class of the variables.

Definition 21 (Reachability). Let $\sigma = \phi \star \mu \in \Sigma_\tau$ and $S \subseteq \Sigma_\tau$. The set of the objects *reachable* in σ is $O_\tau(\sigma) = \cup\{O^i_\tau(\sigma) \mid i \geq 0\}$ where

$$O^0_\tau(S) = \varnothing$$

$$O^{i+1}_\tau(S) = \bigcup \left\{ \{o\} \cup O^i_{F(k(o.\pi))}(o.\phi \star \mu) \;\middle|\; \begin{array}{l} \phi \star \mu \in S,\ v \in \mathsf{dom}(\tau) \\ \phi(v) \in Loc,\ o = \mu\phi(v) \end{array} \right\}.$$

The maps O^i_τ are extended to $\wp(\Sigma_\tau)$ as $O^i_\tau(S) = \cup\{O^i_\tau(\sigma) \mid \sigma \in S\}$.

Proposition 18 provides a guarantee that Definition 21 is well-defined. Observe that variables and fields of type *int* do not contribute to O_τ. We can now define the abstraction map for $\mathcal{E}$. It selects the creation points of the reachable objects.

Definition 22 (Abstraction Map for $\mathcal{E}$). Let $S \subseteq \Sigma_\tau$. The *abstraction map* for $\mathcal{E}$ is

$$\alpha^{\mathcal{E}}_\tau(S) = \{o.\pi \mid \sigma \in S \text{ and } o \in O_\tau(\sigma)\} \subseteq \Pi.$$

Example 23. Let ϕ_1, μ_1, $\sigma_1 = \phi_1 \star \mu_1$, o_1 and o_2 be as defined in Fig. 6. Then $\{v \in \mathsf{dom}(\tau_{w_1}) \mid \phi_1(v) \in Loc\} = \{\mathtt{f}, \mathtt{this}\}$, $\mu_1\phi_1(\mathtt{this}) = o_1$ and $\mu_1\phi_1(\mathtt{f}) = o_2$ so that we have $O^1_\tau(\sigma_1) = \{o_1, o_2\}$. However $o_1.\pi = \overline{\pi}$ and $o_2.\pi = \pi_2$ so that, by using the static information in Fig. 5, we have $F(k(o_1.\pi)) = \varnothing$ and $\mathsf{dom}(F(k(o_2.\pi))) = \{\mathtt{next}, \mathtt{rotation}, \mathtt{side}, \mathtt{Square.x}, \mathtt{Square.y}\}$. From Fig. 6 we conclude that $\{\mu_1(o_2.\phi(f)) \mid f \in \mathsf{dom}(F(k(o_2.\pi))), o_2.\phi(f) \in Loc\} = \{o_2\}$ and therefore

$$O^1_{F(k(o_2.\pi))}(o_2.\phi \star \mu_1) = \left\{ \mu(o_2.\phi(f)) \;\middle|\; \begin{array}{l} f \in \mathsf{dom}(F(k(o_2.\pi))) \\ o_2.\phi(f) \in Loc \end{array} \right\} = \{o_2\}$$

so that $O^2_\tau(\sigma_1) = \{o_1, o_2\} \cup \{o_2\} = \{o_1, o_2\} = O^1_\tau(\sigma_1)$. Thus we have a fixpoint and $O_\tau(\sigma_1) = \{o_1, o_2\}$. Note that $o_4 \notin O_\tau(\sigma_1)$ i.e., it is *garbage*.

As $o_1.\pi = \overline{\pi}$ and $o_2.\pi = \pi_2$, we have $\alpha^{\mathcal{E}}_{\tau_{w_1}}(\sigma_1) = \{\overline{\pi}, \pi_2\}$. This corresponds with the approximation we used in Example 2 to decorate program point w_1.

4.1 The domain $\mathcal{E}$ in the presence of type information

Definition 22 seems to suggest that $\mathsf{rng}(\alpha^{\mathcal{E}}_\tau) = \wp(\Pi)$ i.e., that every set of creation points is a legal approximation in each given program point. However, this is not true if type information is taken into account.

Example 24. Consider the program point w_0 in Fig. 1 and its type environment $\tau_{w_0} = [\mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{Circle}]$. Then $\alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma) \neq \{\pi_4\}$ for every $\sigma = \phi \star \mu \in \Sigma_\tau$. This is because

$$\alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma) = \bigcup \left\{ \{o.\pi\} \bigcup \alpha^{\mathcal{E}}_{F(k(o.\pi))}((o.\phi) \star \mu) \;\middle|\; \begin{array}{l} v \in \{\mathtt{this}\},\ \phi(v) \in Loc \\ o = \mu\phi(v) \end{array} \right\}.$$

By Definition 13 we know that if $\phi(v) \in Loc$ then $k(o.\pi) = \mathtt{Circle}$. Hence $o.\pi = \pi_3$. We conclude that either $\phi(v) = \mathit{null}$ and $\alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma) = \varnothing$, or $\phi(v) \in Loc$ and $\pi_3 \in \alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma)$. In both cases it is not possible that $\alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma) = \{\pi_4\}$.

Example 24 shows that *static type information provides escape information* by indicating which subsets of creation points are not the abstraction of any concrete states. We should therefore characterise which are the *good* or meaningful elements of $\wp(\Pi)$. This is important because it reduces the size of the abstract domain and removes useless creation points during the analysis through the use of an *abstract garbage collector* δ_τ (Definition 25).

Let $e \in \wp(\Pi)$. Then $\delta_\tau(e)$ is defined as the largest subset of e which contains only those creation points deemed useful by the type environment τ. This set is computed first by collecting the creation points that create objects compatible with the types in τ. For each of these points, this check is reiterated for each of the fields of the object it creates until a fixpoint is reached. Note that if there are no possible creation points for `this`, all creation points are useless.

Definition 25 (Abstract Garbage Collector δ). Let $e \subseteq \Pi$. We define $\delta_\tau(e) = \cup\{\delta^i_\tau(e) \mid i \geq 0\}$ with

$$\delta^0_\tau(e) = \varnothing$$

$$\delta^{i+1}_\tau(e) = \begin{cases} \varnothing & \text{if } \texttt{this} \in \mathsf{dom}(\tau) \text{ and no } \pi \in e \text{ is s.t. } k(\pi) \leq \tau(\texttt{this}) \\ \bigcup \Big\{ \{\pi\} \bigcup \delta^i_{F(\pi)}(e) \;\Big|\; \kappa \in \mathsf{rng}(\tau) \bigcap \mathcal{K},\ \pi \in e,\ k(\pi) \leq \kappa \Big\} & \\ \quad \text{otherwise.} & \end{cases}$$

It follows from Definition 25 that $\delta^i_\tau \subseteq \delta^{i+1}_\tau$ and hence $\delta_\tau = \delta^{\#\Pi}_\tau$. Note that in Definition 25 we consider all subclasses of κ (Example 20).

Example 26. Let us look at the program in Fig. 1.

Consider first the program point w_0 and the type environment $\tau_{w_0} = [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{Circle}]$ at program point w_0. Let $e = \{\overline{\pi}, \pi_1, \pi_3, \pi_4\}$. Then, it can be seen in Fig. 5 that $\mathsf{rng}(F(\texttt{Circle})) \cap \mathcal{K} = \{\texttt{Figure}\}$. Note that $\kappa(\pi_3) = \texttt{Circle} = \tau_{w_0}(\texttt{this})$. Thus, for every $i \in \mathbb{N}$, we have

$$\begin{aligned} \delta^1_{F(\texttt{Circle})}(e) &= \cup\Big\{ \{\pi\} \cup \delta^0_{F(k(\pi))}(e) \;\Big|\; \pi \in e,\ k(\pi) \leq \texttt{Figure} \Big\} = \{\pi_3\} \\ \delta^{i+1}_{F(\texttt{Circle})}(e) &= \cup\Big\{ \{\pi\} \cup \delta^i_{F(k(\pi))}(e) \;\Big|\; \pi \in e,\ k(\pi) \leq \texttt{Figure} \Big\} \\ &= \{\pi_3\} \cup \delta^i_{F(\texttt{Circle})(e)}\ , \end{aligned}$$

which is enough to prove, by induction, that $\delta^i_{F(\texttt{Circle})}(e) = \{\pi_3\}$ for every $i \geq 1$. Then we have

$$\begin{aligned} \delta^{i+2}_{\tau_{w_0}}(e) &= \cup\Big\{ \{\pi\} \cup \delta^{i+1}_{F(k(\pi))}(e) \;\Big|\; \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in e,\ k(\pi) \leq \kappa \Big\} \\ &= \cup\Big\{ \{\pi\} \cup \delta^{i+1}_{F(k(\pi))}(e) \;\Big|\; \pi \in e,\ k(\pi) \leq \texttt{Circle} \Big\} \\ &= \{\pi_3\} \cup \delta^{i+1}_{F(\texttt{Circle})}(e) = \{\pi_3\}. \end{aligned}$$

Consider now the program point w_1 and the type environment τ_{w_1} at program point w_1 as given in Fig. 6. Let $e = \{\overline{\pi}, \pi_1, \pi_3, \pi_4\}$. Suppose that $i > 0$. As $\mathsf{rng}(F(\texttt{Scan})) = \varnothing$, we have $\delta^i_{F(\texttt{Scan})}(e) = \varnothing$; in the previous paragraph we have shown that $\delta^i_{F(\texttt{Circle})}(e) = \{\pi_3\}$;

similarly, it can be seen that $\delta^i_{F(\texttt{Square})}(e) = \{\pi_1, \pi_3, \pi_4\}$. We therefore can conclude that, for all $i > 0$,

$$\begin{aligned}
\delta_{\tau_{w_1}}(e) &= \delta^{i+1}_{\tau_{w_1}}(e) \\
&= \cup\left\{\{\pi\} \cup \delta^i_{F(k(\pi))}(e) \;\middle|\; \kappa \in \{\texttt{Figure}, \texttt{Scan}\},\ \pi \in e,\ k(\pi) \leq \kappa\right\} \\
&= \{\overline{\pi}, \pi_3\} \cup \delta^{i+1}_{F(\texttt{Square})}(e) \cup \delta^i_{F(\texttt{Circle})}(e) \cup \delta^i_{F(\texttt{Scan})}(e) \\
&= \{\overline{\pi}, \pi_3\} \cup \{\pi_1, \pi_3, \pi_4\} \cup \{\pi_3\} \cup \varnothing \\
&= \{\overline{\pi}, \pi_1, \pi_3, \pi_4\}.
\end{aligned}$$

Then all the creation points in e are useful in w_1 (compare this with Example 20).

Proposition 27 states that the abstract garbage collector δ_τ is a lower closure operator so that it possesses the properties of monotonicity, reductivity and idempotence that would be expected in a garbage collector.

Proposition 27. *Let $i \in \mathbb{N}$. The abstract garbage collectors δ^i_τ and δ_τ are lco's.*

The following result proves that δ_τ can be used to define $\mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$. Namely, the *useful* elements of $\wp(\Pi)$ are those that do not contain any garbage. The proof of Proposition 28 relies on the explicit construction, for every $e \subseteq \Pi$, of a set of concrete states X such that $\alpha_\tau(X) = \delta_\tau(e)$, which is a fixpoint of δ_τ by a well-known property of lco's.

Proposition 28. *Let $\delta(\tau)$ be an abstract garbage collector. We have that $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$ and $\varnothing \in \mathsf{fp}(\delta_\tau)$. Moreover, if $\texttt{this} \in \mathsf{dom}(\tau)$, then for every $X \subseteq \Sigma_\tau$ we have $\alpha^{\mathcal{E}}_\tau(X) = \varnothing$ if and only if $X = \varnothing$.*

Proposition 28 lets us assume that $\alpha^{\mathcal{E}}_\tau : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\delta_\tau)$. Moreover, it justifies the following definition of our domain $\mathcal{E}$ for escape analysis. Proposition 28 can be used to compute the possible approximations from $\mathcal{E}$ at a given program point. However, it does not specify which of these is best. This is the goal of an escape analysis (Section 4.2).

Definition 29 (Abstract Domain $\mathcal{E}$). Our basic domain for escape analysis is $\mathcal{E}_\tau = \mathsf{fp}(\delta_\tau)$, ordered by set inclusion.

Example 30. Let τ_{w_0} and τ_{w_1} be as given in Fig. 6. Then

$$\begin{aligned}
\mathcal{E}_{\tau_{w_0}} &= \{\varnothing\} \cup \{e \in \wp(\Pi) \mid \pi_3 \in e \text{ and } (\{\pi_1, \pi_4\} \cap e \neq \varnothing \text{ entails } \pi_2 \in e)\} \\
\mathcal{E}_{\tau_{w_1}} &= \{\varnothing\} \cup \{e \in \wp(\Pi) \mid \overline{\pi} \in e \text{ and } (\{\pi_1, \pi_4\} \cap e \neq \varnothing \text{ entails } \pi_2 \in e)\}.
\end{aligned}$$

The constraints say that there must be a creation point for the `this` variable and that to reach an `Angle` (created at π_1 or at π_4) from the variables in $\mathsf{dom}(\tau_{w_0})$ or $\mathsf{dom}(\tau_{w_1})$, we must be able to reach a `Square` (created at π_2).

By Definition 22, we know that $\alpha^{\mathcal{E}}_\tau$ is strict and additive and, by Proposition 28, onto $\mathcal{E}_\tau$. Thus, by a general result of abstract interpretation [14, 15] (Section 2), we have the following proposition.

Proposition 31. *The map $\alpha^{\mathcal{E}}_{\tau}$ (Definition 22) is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{E}_\tau$.*

Note that if, in Definition 29, we had defined $\mathcal{E}_\tau$ as $\wp(\Pi)$, the map $\alpha^{\mathcal{E}}_{\tau}$ would induce just a Galois connection instead of a Galois insertion, as a consequence of Proposition 28.

The domain $\mathcal{E}$ induces optimal abstract operations which can be used for an actual escape analysis. We discuss this in the next subsection.

4.2 Static analysis over $\mathcal{E}$

Figure 9 defines the abstract counterparts of the concrete operations in Fig. 8. Proposition 32 states that they are correct and optimal, in the sense of abstract interpretation (Section 2). Optimality is proved by showing that each operation in Fig. 9 coincides with the optimal operation $\alpha^{\mathcal{E}} \circ op \circ \gamma^{\mathcal{E}}$, where *op* is the corresponding concrete operation in Fig. 8, as required by the abstract interpretation framework. Note that the map $\gamma^{\mathcal{E}}$ is induced by $\alpha^{\mathcal{E}}$ (Section 2).

Proposition 32. *The operations in Fig. 9 are the optimal counterparts induced by $\alpha^{\mathcal{E}}$ of the operations in Fig. 8 and of $\cup$. They are implicitly strict on $\varnothing$, except for* return*, which is strict in its first argument only, and for $\cup$.*

Many operations in Fig. 9 coincide with the identity map. This is a sign of the computational imprecision conveyed by the domain $\mathcal{E}$. Other operations call the δ garbage collector quite often to remove creation points of objects which might become unreachable since some variable has disappeared from the scope. For instance, as the concrete put_var operation removes variable v from the scope (Fig. 8), its abstract counterpart in Fig. 9 calls the garbage collector. The same happens for restrict which, however, removes a *set* of variables from the

$$
\begin{aligned}
\mathsf{nop}_\tau(e) &= e & \mathsf{get\_int}^i_\tau(e) &= e\\
\mathsf{get\_null}^\kappa_\tau(e) &= e & \mathsf{get\_var}^v_\tau(e) &= e\\
\mathsf{is\_true}_\tau(e) &= e & \mathsf{is\_false}_\tau(e) &= e\\
\mathsf{put\_var}^v_\tau(e) &= \delta_{\tau|_{-v}}(e) & \mathsf{is\_null}_\tau(e) &= \delta_{\tau|_{-res}}(e)\\
\mathsf{new}^\pi_\tau(e) &= e \cup \{\pi\} & =_\tau(e_1)(e_2) &= +_\tau(e_1)(e_2) = e_2\\
\mathsf{expand}^{v:t}_\tau(e) &= e & \mathsf{restrict}^{vs}_\tau(e) &= \delta_{\tau_{-vs}}(e)\\
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(e) &= \delta_{\tau|_{\{v_1,\ldots v_n,res\}}}(e) & \cup_\tau(e_1)(e_2) &= e_1 \cup e_2
\end{aligned}
$$

$$
\mathsf{get\_field}^f_\tau(e) = \begin{cases} \varnothing & \text{if } \{\pi \in e \mid k(\pi) \le \tau(res)\} = \varnothing \\ \delta_{\tau[res \mapsto F(\tau(res))(f)]}(e) & \text{otherwise} \end{cases}
$$

$$
\mathsf{put\_field}^f_{\tau,\tau'}(e_1)(e_2) = \begin{cases} \varnothing & \text{if } \{\pi \in e_1 \mid k(\pi) \le \tau(res)\} = \varnothing \\ \delta_{\tau|_{-res}}(e_2) & \text{otherwise} \end{cases}
$$

$$
\mathsf{return}^\nu_\tau(e_1)(e_2) = \cup \left\{ \{\pi\} \cup \delta_{F(k(\pi))}(\Pi) \;\middle|\; \begin{array}{l} \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K} \\ \pi \in e_1,\ k(\pi) \le \kappa \end{array} \right\} \cup e_2
$$

$$
\mathsf{lookup}^{m,\nu}_\tau(e) = \begin{cases} \varnothing & \text{if } e' = \left\{ \pi \in e \;\middle|\; \begin{array}{l} k(\pi) \le \tau(res) \\ M(k(\pi))(m) = \nu \end{array} \right\} = \varnothing \\ \delta_{\tau|_{-res}}(e) \cup \left(\bigcup \{ \{\pi\} \cup \delta_{F(k(\pi))}(e) \mid \pi \in e' \} \right) & \text{otherwise.} \end{cases}
$$

Fig. 9 The optimal abstract operations over $\mathcal{E}$

scope. There are also some operations (is_null, put_field, lookup) that use *res* as a temporary variable and one operation (get_field) that changes the type of *res*. Hence these abstract operations also need to call the garbage collector. Note that the definitions of the get_field, put_field and lookup operations also consider, separately, the unusual situation when we read a field, respectively, write a field or call a method and the receiver is *always null*. In this case, the concrete computation always stops so that the best approximation of the (empty) set of subsequent states is $\varnothing$. The garbage collector is also called by call since it creates a scope for the callee where only some of the variables of the caller (namely, the parameters of the callee) are addressable. The new operation adds its creation point to the approximation, since its concrete counterpart creates an object and binds it to the temporary variable *res*. The $\cup$ operation computes the union of the creation points reachable from at least one of the two branches of a conditional. The return operation states that all fields of the objects bound to the variables in scope before the call might have been modified by the call. This is reflected by the use of $\delta_{F(k(\pi))}(\Pi)$ in return, which plays the role of a worst-case assumption on the content of the fields. After Example 33 we discuss how to cope with the possible imprecision of this definition. The lookup operation computes first the set e' of the creation points of objects that may be receivers of the virtual call. If this set is not empty, the variable *res* (which holds the receiver of the call) is required to be bound to an object created at some creation point in e'. This further constrains the creation points reachable from *res* and this is why we call the garbage collector $\delta_{F(k(\pi))}$ for each $\pi \in e'$.

The definitions of return and lookup are quite complex; this is a consequence of our quest for *optimal* abstract operations. It is possible to replace their definitions in Fig. 9 by the less precise but simpler definitions:

$$\mathsf{return}^{v}_{\tau}(e_1)(e_2) = \delta_\tau(\Pi) \cup e_2 \qquad \mathsf{lookup}^{m,v}_{\tau}(e) = e.$$

Note though that, in practice, the results with the simpler definitions will often be the same.

Example 33. Let us mimic, in $\mathcal{E}$, the concrete computation of Example 19. Let the type environment $\tau = \tau_{w_1}$ and the concrete states σ_1 and σ_2 be as given in Fig. 6. We start by constructing elements e_1 and e_2 of $\mathcal{E}$ corresponding to σ_1 and σ_2. The abstract state e_1 is obtained by abstracting σ_1 (see Example 23):

$$e_1 = \alpha^{\mathcal{E}}_{\tau}(\{\sigma_1\}) = \alpha^{\mathcal{E}}_{\tau}(\sigma_1) = \{\overline{\pi}, \pi_2\}$$
$$e_2 = \mathsf{get\_var}^{\mathsf{f}}_{\tau}(e_1) = \{\overline{\pi}, \pi_2\}.$$

There are three abstract lookup operations corresponding to the concrete ones and hence we construct for $i = 3, \ldots, 6$, elements e'_i, e''_i, e'''_i of $\mathcal{E}$ corresponding to the concrete states σ'_i, σ''_i, σ'''_i, respectively.

$$e'_3 = \mathsf{lookup}^{\mathtt{def},\mathtt{Figure.def}}_{\tau[res\mapsto \mathtt{Figure}]}(e_2) = \varnothing$$
$$\text{since } \{\pi \in e_2 \mid k(\pi) \leq \mathtt{Figure} \text{ and } M(\pi)(\mathtt{def}) = \mathtt{Figure.def}\} = \varnothing$$
$$e''_3 = \mathsf{lookup}^{\mathtt{def},\mathtt{Square.def}}_{\tau[res\mapsto \mathtt{Figure}]}(e_2)$$
$$= \delta_\tau(e_2) \cup \left(\bigcup \left\{ \{\pi\} \cup \delta_{F(k(\pi))}(e_2) \,\middle|\, \pi \in e'' \right\}\right) = \{\overline{\pi}, \pi_2\} \cup \{\pi_2\}$$
$$= \{\overline{\pi}, \pi_2\},$$

$$\text{since } e'' = \left\{ \pi \in e_2 \;\middle|\; \begin{array}{l} k(\pi) \leq \texttt{Figure} \\ M(\pi)(\texttt{def}) = \texttt{Square.def} \end{array} \right\} = \{\pi_2\}$$

$$e_3''' = \mathsf{lookup}^{\texttt{def},\texttt{Circle.def}}_{\tau[res \mapsto \texttt{Figure}]}(e_2) = \varnothing$$

$$\text{since } e''' = \left\{ \pi \in e_2 \;\middle|\; \begin{array}{l} k(\pi) \leq \texttt{Figure} \\ M(\pi)(\texttt{def}) = \texttt{Circle.def} \end{array} \right\} = \varnothing.$$

The lookup operations for `Figure` and `Circle` return $\varnothing$ so that, as the abstract operations over $\mathcal{E}$ are strict on $\varnothing$ (Proposition 32), $e_4' = e_5' = e_6' = e_4''' = e_5''' = e_6''' = \varnothing$. This is because the analysis is able to guess the target of the virtual call `f.def()`, since the only objects reachable there are a `Square` (created in π_2) and a `Scan` which, however, is not compatible with the declared type of the receiver of the call. Hence we only have to consider the case when a `Square` is selected:

$$e_4'' = \mathsf{call}^{\texttt{Square.def}}_{\tau[res \mapsto \texttt{Square}]}(e_3'') = \delta_{[res \mapsto \texttt{Square}]}(e_3'') = \{\pi_2\}.$$

Since $P(\texttt{Square.def})|_{\texttt{out}} = [\texttt{out} \mapsto \mathit{int}]$ and $\mathcal{E}_{[\texttt{out} \mapsto \mathit{int}]} = \{\varnothing\}$, we do not need to execute the method `Square.def` to conclude that

$$e_5'' = \varnothing.$$

Hence

$$\begin{aligned} e_6'' &= \mathsf{return}^{\texttt{Square.def}}_{\tau[res \mapsto \texttt{Square}]}(e_3'')(e_5'') \\ &= \cup\{\{\pi\} \cup \delta_{F(k(\pi))}(\Pi) \mid \pi \in \{\overline{\pi}, \pi_2\}\} = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}. \end{aligned}$$

Since the abstract semantics is non-deterministic, we merge the results of every thread of execution through the $\cup$ operation. Hence the abstract state after the execution of the call `f.def()` in Fig. 1 is

$$e_6 = e_6' \cup e_6'' \cup e_6''' = \varnothing \cup \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\} \cup \varnothing = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}.$$

In Example 33, the imprecision of the analysis induced by $\mathcal{E}$ is largely due to the abstract operation return used to compute e_6''. The creation points π_1 and π_4 for `Angle`s need to be added because in the execution of the methods `Square.def` any field of the object bound to `this` *could* be modified. For instance, an `Angle` could be bound to the field `rotation` of the object bound to `this`. This is what actually happens for π_1 in the method `Square.def` (Fig. 1), while the introduction of the creation point π_4 is an imprecise (but correct) assumption. This is a consequence of the definition of $\mathcal{E}$ as the set of sets of *reachable* creation points. At the end of a method, we only have rather weak information about the set e of creation points of the objects reachable from `out`. For instance, if `out` has type *int*, such as in Example 33, we can only have $e = \varnothing$. When we return to the caller, the actual parameters return into scope. The definition of return in Fig. 8 shows that these parameters are unchanged (because of the condition $\mu_1 =_L \mu_2$, where $L = \mathsf{rng}(\phi_1)|_{-res} \cap \mathit{Loc}$, on the concrete operation). This is reflected by the condition $\pi \in e_1$ in the abstract return operation in Fig. 9. However, we do not know anything about their *fields*. Without such information, only a pessimistic assumption can be made, which is expressed by the use of Π in the abstract return operation. This problem can be solved by including a *shadow copy* of the actual parameters among the variables in scope inside a method. An example of the use of this technique will be given

later (see Example 52). By using this technique, we can actually improve the precision of the computation in Example 33. As reported in Example 2, we get the more precise approximation $\{\overline{\pi}, \pi_1, \pi_2\}$ after the first call to `f.def()`.

There is, however, another problem related with the domain $\mathcal{E}$. It is exemplified below.

Example 34. Let the approximation provided by $\mathcal{E}$ before the statement `f = new Circle()` in Fig. 1 be $\{\overline{\pi}, \pi_1, \pi_2, \pi_4\}$ (Example 2). We can compute the approximation *after* that statement by executing two abstract operations. Since the object created in π_3 gets stored inside the variable `f`, we would expect the creation point π_2 of the old value of `f` to disappear. But this does not happen:

$$\mathsf{new}_{\tau}^{\pi_3}(\{\overline{\pi}, \pi_1, \pi_2, \pi_4\}) = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$$

$$\mathsf{put\_var}^{\mathtt{f}}_{\tau[res \mapsto \mathtt{Circle}]}(\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}) = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}.$$

This time, the imprecision is a consequence of the fact that $\mathtt{n} \in \mathsf{dom}(\tau)$, $\tau(\mathtt{n}) = \mathtt{Figure}$ and $k(\pi_2) = \mathtt{Square} \leq \mathtt{Figure}$. Since the abstract domain $\mathcal{E}$ does not allow one to know the creation points of the objects bound to *a given* variable, but only provides global information on the creation points of the objects bound to variables and fields *as a whole*, we do not know whether π_2 is the creation point of an object bound to `f` (and in such a case it disappears) or to `n` instead (and in such a case it must *not* disappear). Hence a correct put_var operation cannot make π_2 disappear. We solve this problem in Section 5 by introducing this missing information into a new, more precise, abstract domain $\mathcal{ER}$.

5 The refined domain $\mathcal{ER}$

We define here a *refinement* $\mathcal{ER}$ of the domain $\mathcal{E}$ of Section 4, in the sense that $\mathcal{ER}$ is a concretisation of $\mathcal{E}$ (Proposition 56). The idea underlying the definition of $\mathcal{ER}$ is that the precision of $\mathcal{E}$ can be improved if we can speak about the creation points of the objects bound to *a given* variable or field (see the problem highlighted in Example 34). The construction of $\mathcal{ER}$ is very similar to that of $\mathcal{E}$.

5.1 The domain

Definition 11 defines concrete values. The domain $\mathcal{ER}$ we are going to define approximates every concrete value with an *abstract value*. An abstract value is either $*$, which approximates the integers, or a set $e \subseteq \Pi$, which approximates *null* and all locations containing an object created in some creation point in e. An abstract frame maps variables to abstract values consistent with their type.

Definition 35 (Abstract Values and Frames). Let the *abstract values* be $\mathit{Value}^{\mathcal{ER}} = \{*\} \cup \wp(\Pi)$. We define

$$\mathit{Frame}_{\tau}^{\mathcal{ER}} = \left\{ \phi \in \mathsf{dom}(\tau) \mapsto \mathit{Value}^{\mathcal{ER}} \;\middle|\; \begin{array}{l} \text{for every } v \in \mathsf{dom}(\tau) \\ \text{if } \tau(v) = \mathit{int} \text{ then } \phi(v) = * \\ \text{if } \tau(v) \in \mathcal{K} \text{ and } \pi \in \phi(v) \\ \quad \text{then } k(\pi) \leq \tau(v) \end{array} \right\}.$$

The set $\mathit{Frame}_{\tau}^{\mathcal{ER}}$ is ordered by pointwise set-inclusion.

Example 36. Let τ_{w_1} be as defined in Fig. 6. Then we have

$$[\mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \{\pi_2, \pi_3\}, \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\}] \in \mathit{Frame}^{\mathcal{ER}}_{\tau_{w_1}}$$

$$[\mathtt{f} \mapsto \{\overline{\pi}, \pi_2\}, \mathtt{n} \mapsto \{\pi_2, \pi_3\}, \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\}] \notin \mathit{Frame}^{\mathcal{ER}}_{\tau_{w_1}},$$

since $k(\overline{\pi}) = \mathtt{Scan}$, $\tau_{w_1}(\mathtt{f}) = \mathtt{Figure}$ and $\mathtt{Scan} \not\leq \mathtt{Figure}$.

The map ε *extracts* the creation points of the objects bound to the variables.

Definition 37 (Extraction Map). The map $\varepsilon_\tau : \wp(\Sigma_\tau) \mapsto \mathit{Frame}^{\mathcal{ER}}_\tau$ is such that, for every $S \subseteq \Sigma_\tau$ and $v \in \mathsf{dom}(\tau)$,

$$\varepsilon_\tau(S)(v) = \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{(\mu\phi(v)).\pi \mid \phi \star \mu \in S \text{ and } \phi(v) \in \mathit{Loc}\} & \text{if } \tau(v) \in \mathcal{K}. \end{cases}$$

Example 38. Consider the state σ_1 in Fig. 6. Then

$$\varepsilon_{\tau_{w_1}}(\sigma_1) = [\mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing, \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\}].$$

Since it is assumed that all the fields are uniquely identified by their fully qualified name, the type environment $\widetilde{\tau}$ *of all the fields* introduced by the program is well-defined.

Definition 39 (Type Environment of All Fields).We define the *type environment of all fields* as $\widetilde{\tau} = \cup\{F(\kappa) \mid \kappa \in \mathcal{K}\}$. Let $\tau \in \mathit{TypEnv}$ be such that $\mathsf{dom}(\tau) \subseteq \mathsf{dom}(\widetilde{\tau})$ and $\phi \in \mathit{Frame}_\tau$. Its *extension* $\widetilde{\phi} \in \mathit{Frame}_{\widetilde{\tau}}$ is such that, for every $v \in \mathsf{dom}(\widetilde{\tau})$,

$$\widetilde{\phi}(v) = \begin{cases} \phi(v) & \text{if } v \in \mathsf{dom}(\tau) \\ \Im(\widetilde{\tau}(v)) & \text{otherwise (Definition 11).} \end{cases}$$

Example 40. Consider the map F in Fig. 5 for the program in Fig. 1. Then

$$\widetilde{\tau} = \begin{bmatrix} \mathtt{Circle.x} \mapsto \mathit{int}, \mathtt{Circle.y} \mapsto \mathit{int}, \mathtt{degree} \mapsto \mathit{int} \\ \mathtt{next} \mapsto \mathtt{Figure}, \mathtt{radius} \mapsto \mathit{int}, \mathtt{rotation} \mapsto \mathtt{Angle} \\ \mathtt{side} \mapsto \mathit{int}, \mathtt{Square.x} \mapsto \mathit{int}, \mathtt{Square.y} \mapsto \mathit{int} \end{bmatrix}.$$

Let $\phi = [\mathtt{Circle.x} \mapsto 12, \mathtt{Circle.y} \mapsto 5, \mathtt{next} \mapsto l, \mathtt{radius} \mapsto 5] \in F(\mathtt{Circle})$, with $l \in \mathit{Loc}$. We have

$$\widetilde{\phi} = \begin{bmatrix} \mathtt{Circle.x} \mapsto 12, \mathtt{Circle.y} \mapsto 5, \mathtt{degree} \mapsto 0 \\ \mathtt{next} \mapsto l, \mathtt{radius} \mapsto 5, \mathtt{rotation} \mapsto \mathit{null}, \mathtt{side} \mapsto 0 \\ \mathtt{Square.x} \mapsto 0, \mathtt{Square.y} \mapsto 0 \end{bmatrix}.$$

An abstract memory is an abstract frame for $\widetilde{\tau}$. The abstraction map computes the abstract memory by extracting the creation points of the fields of the reachable objects of the concrete memory (Definition 21).

Definition 41 (Abstract Map for $\mathcal{ER}$). Let the set of abstract memories be $Memory^{\mathcal{ER}} = Frame^{\mathcal{ER}}_{\tau}$. We define the map

$$\alpha^{\mathcal{ER}}_{\tau} : \wp(\Sigma_\tau) \mapsto \{\bot\} \cup \left(Frame^{\mathcal{ER}}_{\tau} \times Memory^{\mathcal{ER}}\right)$$

such that, for $S \subseteq \Sigma_\tau$,

$$\alpha^{\mathcal{ER}}_{\tau}(S) = \begin{cases} \bot & \text{if } S = \varnothing \\ \varepsilon_\tau(S) \star \varepsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi} \star \sigma.\mu \mid \sigma \in S \text{ and } o \in O_\tau(\sigma)\}) & \text{otherwise.} \end{cases}$$

Example 42. Consider the state σ_1 in Fig. 6. Let $\tau = \tau_{w_1}$ be as given in Fig. 6. In Example 23 we have shown that $O_\tau(\sigma_1) = \{o_1, o_2\}$ and in Example 38 we have computed the value of $\varepsilon_\tau(\sigma_1)$. We have

$$\widetilde{o_1.\phi} = \begin{bmatrix} \texttt{Circle.x} \mapsto 0, \texttt{Circle.y} \mapsto 0, \texttt{degree} \mapsto 0 \\ \texttt{next} \mapsto \mathit{null}, \texttt{radius} \mapsto 0, \texttt{rotation} \mapsto \mathit{null} \\ \texttt{side} \mapsto 0, \texttt{Square.x} \mapsto 0, \texttt{Square.y} \mapsto 0 \end{bmatrix},$$

$$\widetilde{o_2.\phi} = \begin{bmatrix} \texttt{Circle.x} \mapsto 0, \texttt{Circle.y} \mapsto 0, \texttt{degree} \mapsto 0 \\ \texttt{next} \mapsto l', \texttt{radius} \mapsto 0, \texttt{rotation} \mapsto \mathit{null} \\ \texttt{side} \mapsto 4, \texttt{Square.x} \mapsto 3, \texttt{Square.y} \mapsto -5 \end{bmatrix}$$

Then (fields not represented are implicitly bound to $*$)

$$\begin{aligned} \alpha^{\mathcal{ER}}_{\tau}(\sigma_1) &= \alpha^{\mathcal{ER}}_{\tau}(\phi_1 \star \mu_1) \\ &= \varepsilon_\tau(\sigma_1) \star \varepsilon_{\widetilde{\tau}}(\{\widetilde{o_1.\phi} \star \mu_1, \widetilde{o_2.\phi} \star \mu_1\}) \\ &= \varepsilon_\tau(\sigma_1) \star \varepsilon_{\widetilde{\tau}}(\{\widetilde{o_2.\phi} \star \mu_1\}) \\ &= \varepsilon_\tau(\sigma_1) \star [\texttt{next} \mapsto \{\mu_1(l').\pi\}, \texttt{rotation} \mapsto \varnothing, \ldots] \\ &= [\texttt{f} \mapsto \{\pi_2\}, \texttt{n} \mapsto \varnothing, \texttt{out} \mapsto *, \texttt{this} \mapsto \{\overline{\pi}\}] \\ &\qquad \star [\texttt{next} \mapsto \{\pi_2\}, \texttt{rotation} \mapsto \varnothing, \ldots]. \end{aligned}$$

Compare Examples 42 and 23. You can see that $\mathcal{ER}$ distributes over the variables and fields the same creation points observed by $\mathcal{E}$.

As a notational simplification, we often assume that each field not reported in the approximation of the memory is implicitly bound to $\varnothing$, if it has class type, and bound to $*$, if it has *int* type.

Just as for $\alpha^{\mathcal{E}}_{\tau}$ (Example 24), the following example shows that the map $\alpha^{\mathcal{ER}}_{\tau}$ is not necessarily onto.

Example 43. Let $\tau = [\texttt{c} \mapsto \texttt{Circle}]$. A `Circle` has no field called `rotation`. Then there is no state $\sigma \in \Sigma_\tau$ such that its abstraction is $\alpha^{\mathcal{ER}}_{\tau}(\sigma) = [\texttt{c} \mapsto \{\pi_3\}] \star [\texttt{next} \mapsto \varnothing, \texttt{rotation} \mapsto \{\pi_1\}, \ldots]$, since only a `Circle` created at π_3 is reachable from the variables.

Hence, we define a map ξ which forces to $\varnothing$ the fields of type class of the objects which have no reachable creation points. Just as for the garbage collector δ for $\mathcal{E}$, the map ξ can be seen

as an abstract garbage collector for $\mathcal{ER}$. This ξ uses an auxiliary map ρ to compute the set of creation points r reachable from the variables in scope. The approximations of the fields of the objects created at r are not garbage collected by ξ. The approximations of the other fields are garbage collected instead.

Definition 44 (Abstract Garbage Collector ξ). We define $\rho_\tau : \mathit{Frame}_\tau^{\mathcal{ER}} \times \mathit{Memory}^{\mathcal{ER}} \mapsto \wp(\Pi)$ and $\xi_\tau : \{\bot\} \cup (\mathit{Frame}_\tau^{\mathcal{ER}} \times \mathit{Memory}^{\mathcal{ER}}) \mapsto \{\bot\} \cup (\mathit{Frame}_\tau^{\mathcal{ER}} \times \mathit{Memory}^{\mathcal{ER}})$ as $\rho_\tau(s) = \cup\{\rho_\tau^i(s) \mid i \geq 0\}$, where

$$\rho_\tau^0(\phi \star \mu) = \varnothing$$

$$\rho_\tau^{i+1}(\phi \star \mu) = \bigcup \left\{ \{\pi\} \cup \rho^i_{F(k(\pi))}(\mu|_{\mathsf{dom}(F(k(\pi)))} \star \mu) \middle| \begin{array}{l} v \in \mathsf{dom}(\tau) \\ \pi \in \phi(v) \end{array} \right\}$$

and

$$\xi_\tau(\bot) = \bot$$

$$\xi_\tau(\phi \star \mu) = \begin{cases} \bot & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and } \phi(\mathtt{this}) = \varnothing \\ \phi \star \left(\cup\{\mu|_{\mathsf{dom}(F(k(\pi)))} \mid \pi \in \rho_\tau(\phi \star \mu)\}\right) & \text{otherwise.} \end{cases}$$

Example 45. Let $s = [\mathtt{c} \mapsto \{\pi_3\}] \star [\mathtt{next} \mapsto \varnothing, \mathtt{rotation} \mapsto \{\pi_1\}, \ldots]$. We have $\rho_\tau(s) = \{\pi_3\}$ and hence

$$\xi_\tau(s) = [\mathtt{c} \mapsto \{\pi_3\}] \star [\mathtt{next} \mapsto \varnothing, \mathtt{rotation} \mapsto \varnothing, \ldots]$$

i.e., the abstract garbage collector ξ has recognised π_1 as garbage (compare with Example 43).

The following property is expected to hold for a garbage collector. Compare Propositions 27 and 46.

Proposition 46. *The abstract garbage collector ξ_τ is an lco.*

The garbage collector ξ_τ can be used to define $\mathsf{rng}(\alpha_\tau^{\mathcal{ER}})$. Namely, the *useful* elements of $\mathit{Frame}_\tau^{\mathcal{ER}} \times \mathit{Memory}^{\mathcal{ER}}$ are exactly those that do not contain any garbage. Compare Propositions 28 and 47.

Proposition 47. *Let ξ_τ be the abstract garbage collector of Definition* 44. *Then* $\mathsf{fp}(\xi_\tau) = \mathsf{rng}(\alpha_\tau^{\mathcal{ER}})$.

Proposition 47 allows us to assume that $\alpha_\tau^{\mathcal{ER}} : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\xi_\tau)$ and justifies the following definition.

Definition 48 (Abstract Domain $\mathcal{ER}$). We define $\mathcal{ER}_\tau = \mathsf{fp}(\xi_\tau)$, ordered by pointwise set-inclusion (with the assumption that $* \subseteq *$ and $\bot \subseteq s$ for every $s \in \mathcal{ER}_\tau$).

By Definitions 37 and 41 we know that the map $\alpha_\tau^{\mathcal{ER}}$ is strict and additive. By Proposition 47 we know that it is onto. Thus we have the following result corresponding to Proposition 31 for the domain $\mathcal{E}$.

Proposition 49. *The map $\alpha_\tau^{\mathcal{ER}}$ is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{ER}_\tau$.*

5.2 Static analysis over $\mathcal{ER}$

In order to use the domain $\mathcal{ER}$ for an escape analysis, we need to provide the abstract counterparts over $\mathcal{ER}$ of the concrete operations in Fig. 8. Since $\mathcal{ER}$ approximates every variable and field with an abstract value, those abstract operations are similar to those of the Palsberg and Schwartzbach's domain for *class analysis* in [35] as formulated in [45]. However, $\mathcal{ER}$ observes the fields of just the reachable objects (Definition 41), while Palsberg and Schwartzbach's domain observes the fields of all objects in memory.

Figure 10 reports the abstract counterparts on $\mathcal{ER}$ of the concrete operations in Fig. 8. These operations are implicitly strict on $\bot$ except for $\cup$. In this case, we define $\bot \cup (\phi \star \mu) = (\phi \star \mu) \cup \bot = \phi \star \mu$. Their optimality is proved by showing that each operation in Fig. 10 coincides with the optimal operation $\alpha^{\mathcal{ER}} \circ op \circ \gamma^{\mathcal{ER}}$, where *op* is the corresponding concrete operation in Fig. 8, as required by the abstract interpretation framework. Note that the map $\gamma^{\mathcal{ER}}$ is induced by $\alpha^{\mathcal{ER}}$ (Section 2).

Proposition 50. *The operations in Fig.* 10 *are the optimal counterparts induced by $\alpha^{\mathcal{ER}}$ of the operations in Fig.* 8 *and of $\cup$.*

Let us consider each of the abstract operations. The operation nop leaves the state unchanged. The same happens for the operations working with integer values only, such as is_true, is_false, $=$ and $+$, since the domain $\mathcal{ER}$ ignores variables with integer values. The concrete operation get_int loads an integer into *res*. Hence, its abstract counterpart loads $*$ into *res*, since $*$ is the approximation for integer values (Definition 35). The concrete operation get_null loads *null* into *res* and hence its abstract counterpart approximates *res* with $\varnothing$. The operation $\mathsf{get\_var}^v$ copies the creation points of v into those of *res*. The $\cup$ operation merges the creation points of the objects bound to each given variable or field in one of the two branches of a conditional. The concrete is_null operation checks if *res* contains *null* or not, and loads 1 or -1 in *res* accordingly. Hence its abstract counterpart loads $*$ into *res*. Since the old value of *res* may no longer be reachable, we apply the abstract garbage collector ξ. The new^π operation binds *res* to an object created at π. The $\mathsf{put\_var}^v$ operation copies the value of *res* into v, and removes *res*. Since the old value of v may be lost, we apply the abstract garbage collector ξ. The restrict operation removes some variables from the scope and, hence, calls ξ. The expand^v operation adds the variable v in scope. Its initial value is approximated with $*$, if it is 0, and with $\varnothing$, if it is *null*. The $\mathsf{get\_field}^f$ operation returns $\bot$ if it is *always* applied to states where the receiver *res* is *null*. This is because $\bot$ is the best approximation of the empty set of final states. If, instead, the receiver is not necessarily *null*, the creation points of the field f are copied from the approximation $\mu(f)$ into the approximation of *res*. Since this operation changes the value of *res*, possibly making some object unreachable, it needs to call ξ. For the $\mathsf{put\_field}^f$ operation, we first check if the receiver is always *null*, in which case the abstract operation returns $\bot$. Then we consider the case in which the evaluation of what is going to be put inside the field makes the receiver unreachable. This (pathological) case happens in a situation such as `a.g.f = m(a)` where the method call `m(a)` sets to *null* the field `g` of the object bound to `a`. Since we assume that the left-hand side is evaluated before the right-hand side, the receiver is not necessarily *null*, but the field updates might not be observable if `a.g.f` is only reachable from `a`. In the third and final case for put_field we consider the standard

$$
\begin{aligned}
\mathsf{nop}_\tau(\phi\star\mu) &= \phi\star\mu\\
\mathsf{get\_int}^i_\tau(\phi\star\mu) &= \phi[res\mapsto *]\star\mu\\
\mathsf{get\_null}^\kappa_\tau(\phi\star\mu) &= \phi[res\mapsto \varnothing]\star\mu\\
\mathsf{get\_var}^v_\tau(\phi\star\mu) &= \phi[res\mapsto \phi(v)]\star\mu\\
\mathsf{is\_true}_\tau(\phi\star\mu) &= \phi\star\mu\\
\mathsf{is\_false}_\tau(\phi\star\mu) &= \phi\star\mu\\
\cup_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) &= (\phi_1\cup\phi_2)\star(\mu_1\cup\mu_2)\\
\mathsf{is\_null}_\tau(\phi\star\mu) &= \xi_{\tau[res\mapsto int]}(\phi[res\mapsto *]\star\mu)\\
\mathsf{new}^\pi_\tau(\phi\star\mu) &= \phi[res\mapsto\{\pi\}]\star\mu\\
\mathsf{put\_var}^v_\tau(\phi\star\mu) &= \xi_{\tau|_{-res}}(\phi[v\mapsto\phi(res)]|_{-res}\star\mu)\\
\mathsf{restrict}^{vs}_\tau(\phi\star\mu) &= \xi_{\tau|_{-vs}}(\phi|_{-vs}\star\mu)\\
\mathsf{expand}^{v:t}_\tau(\phi\star\mu) &= \begin{cases}\phi[v\mapsto *]\star\mu & \text{if } t = int\\ \phi[v\mapsto\varnothing]\star\mu & \text{otherwise}\end{cases}\\
=_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) &= +_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = \phi_2\star\mu_2
\end{aligned}
$$

$$
\mathsf{get\_field}^f_\tau(\phi\star\mu) = \begin{cases}\bot & \text{if } \phi(res)=\varnothing\\ \xi_{\tau[res\mapsto F(\tau(res))(f)]}(\phi[res\mapsto\mu(f)]\star\mu) & \text{else}\end{cases}
$$

$$
\begin{array}{c}\mathsf{put\_field}^f_{\tau,\tau'}\\ (\phi_1\star\mu_1)(\phi_2\star\mu_2)\end{array} = \begin{cases}\bot & \text{if } \phi_1(res)=\varnothing\\ \xi_{\tau|_{-res}}(\phi_2|_{-res}\star\mu_2) & \\ \quad\text{else, if no } \pi\in\phi_1(res) \text{ occurs in } \phi_2|_{-res}\star\mu_2 & \\ \xi_{\tau|_{-res}}(\phi_2|_{-res}\star\mu_2[f\mapsto\mu_2(f)\cup\phi_2(res)]) & \\ \quad\text{otherwise} &\end{cases}
$$

$$
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(\phi\star\mu) = \xi_{P(\nu)|_{-\mathtt{out}}}\left(\left[\begin{array}{l}\iota_1\mapsto\phi(v_1),\ldots,\iota_n\mapsto\phi(v_n)\\ \mathtt{this}\mapsto\phi(res)\end{array}\right]\star\mu\right)
$$

$$
\begin{array}{c}\mathsf{return}^\nu_\tau\\ (\phi_1\star\mu_1)(\phi_2\star\mu_2)\end{array} = \xi_{\tau|_{-res}}(\phi_1|_{-res}\star\mu^\top)\cup([res\mapsto\phi_2(\mathtt{out})]\star\mu_2)
$$

where $\mu^\top$ is the top of $Memory^{\mathcal{ER}}$

$$
\mathsf{lookup}^{m,\nu}_\tau(\phi\star\mu) = \begin{cases}\bot & \text{if } e=\{\pi\in\phi(res)\mid M(\pi)(m)=\nu\}=\varnothing\\ \xi_\tau(\phi[res\mapsto e]\star\mu) & \text{otherwise.}\end{cases}
$$

Fig. 10 The abstract operations over $\mathcal{ER}$

situation when we write into a reachable field of a non-*null* receiver. The creation points of the right-hand side are added to those already approximating the objects stored in f. The call operation restricts the scope to the parameters passed to a method and hence ξ is used. The return operation copies into *res* the return value of the method which is held in `out`. The local variables of the caller are put back into scope, but the approximation of their fields is provided through a worst-case assumption $\mu^\top$ since they may be modified by the call. This loss of precision can be overcome by means of shadow copies of the variables, just as for $\mathcal{E}$ (see Example 52). The lookup^m operation first computes the subset e of the approximation of the receiver of the call only containing the creation points whose class leads to a call to the

method m. If $e = \varnothing$, a call to m is impossible and the result of the operation is $\bot$. Otherwise, e becomes the approximation of the receiver *res*, so that some creation points can disappear and we need to call ξ.

Example 51. As in Example 33 for $\mathcal{E}$, let us mimic, in $\mathcal{ER}$, the concrete computation of Example 19. We start from the abstraction (Definition 41) of σ_1, given in Example 42. Variables and fields not shown are implicitly bound to $\varnothing$ if they have class type and to $*$ if they have type *int*.

$$s_1 = \alpha_\tau^{\mathcal{ER}}(\sigma_1) = \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing \\ \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \varnothing \end{bmatrix}$$

$$s_2 = \mathsf{get\_var}_\tau^{\mathtt{f}}(s_1) = \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing \\ \mathit{res} \mapsto \{\pi_2\}, \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \varnothing \end{bmatrix}.$$

There are three abstract lookup operations corresponding to the concrete ones and hence we construct for $i = 3, \ldots, 6$, elements s_i', s_i'', s_i''' of $\mathcal{ER}$ corresponding to the concrete states σ_i', σ_i'', σ_i''', respectively.

$$s_3' = \mathsf{lookup}_{\tau[\mathit{res}\mapsto\mathtt{Figure}]}^{\mathtt{def},\mathtt{Figure.def}}(s_2) = \bot$$
$$\text{since } e' = \{\pi \in \{\pi_2\} \mid M(\pi)(\mathtt{def}) = \mathtt{Figure.def}\} = \varnothing,$$
$$s_3'' = \mathsf{lookup}_{\tau[\mathit{res}\mapsto\mathtt{Figure}]}^{\mathtt{def},\mathtt{Square.def}}(s_2) = \xi_\tau(s_2) = s_2$$
$$\text{since } e'' = \{\pi \in \{\pi_2\} \mid M(\pi)(\mathtt{def}) = \mathtt{Square.def}\} = \{\pi_2\},$$
$$s_3''' = \mathsf{lookup}_{\tau[\mathit{res}\mapsto\mathtt{Figure}]}^{\mathtt{def},\mathtt{Circle.def}}(s_2) = \bot$$
$$\text{since } e''' = \{\pi \in \{\pi_2\} \mid M(\pi)(\mathtt{def}) = \mathtt{Circle.def}\} = \varnothing.$$

The lookup operations for `Figure` and `Circle` return $\bot$ so that, as the abstract operations over $\mathcal{ER}$ are strict on $\bot$ (Proposition 50), $s_4' = s_5' = s_6' = s_4''' = s_5''' = s_6''' = \bot$. This is because the analysis is able to guess the target of the virtual call `f.def()`, since the only creation point for the receiver `f` is π_2, which creates `Square`s. Hence we only have to consider the case when a `Square` is selected:

$$\begin{aligned} s_4'' &= \mathsf{call}_{\tau[\mathit{res}\mapsto\mathtt{Square}]}^{\mathtt{Square.def}}(s_3'') \\ &= \xi_{[\mathit{res}\mapsto\mathtt{Square}]}\left([\mathtt{this} \mapsto \{\pi_2\}] \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \varnothing \end{bmatrix}\right) \\ &= [\mathtt{this} \mapsto \{\pi_2\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \varnothing]. \end{aligned}$$

Since $P(\mathtt{Square.def})|_{\mathtt{out}} = [\mathtt{out} \mapsto \mathit{int}]$ and $\mathcal{ER}_{[\mathtt{out}\mapsto\mathit{int}]} = \{\bot, [\mathtt{out} \mapsto *] \star [\mathtt{next} \mapsto \varnothing,\ \mathtt{rotation} \mapsto \varnothing]\}$, we can just observe that the method `Square.def` does not diverge to conclude that

$$s_5'' = [\mathtt{out} \mapsto *] \star [\mathtt{next} \mapsto \varnothing,\ \mathtt{rotation} \mapsto \varnothing].$$

Hence, by letting $\mu^\top$ denote the top element of $Memory^{\mathcal{ER}}$ so that

$$\mu^\top = [\mathtt{next} \mapsto \{\pi_2, \pi_3\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}],$$

we have

$$\begin{aligned} s_6'' &= \mathsf{return}^{\mathtt{Square.def}}_{\tau[res \mapsto \mathtt{Square}]}(s_3'')(s_5'') \\ &= \mathsf{return}^{\mathtt{Square.def}}_{\tau[res \mapsto \mathtt{Square}]}(s_2)(s_5'') \\ &= \xi_{\left[\begin{smallmatrix}\mathtt{f}\mapsto\mathtt{Figure},\ \mathtt{n}\mapsto\mathtt{Figure},\\ \mathtt{out}\mapsto int,\ \mathtt{this}\mapsto\mathtt{Scan}\end{smallmatrix}\right]}([\mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing, \mathtt{this} \mapsto \{\overline{\pi}\}] \star \mu^\top) \\ &\qquad \cup ([res \mapsto *] \star [\mathtt{next} \mapsto \varnothing,\ \mathtt{rotation} \mapsto \varnothing]) \\ &= [\mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing, \mathtt{this} \mapsto \{\overline{\pi}\}] \star \mu^\top. \end{aligned}$$

Since the abstract semantics is non-deterministic, we merge the results of every thread of execution through the $\cup$ operation. Hence the abstract state after the execution of the call `f.def()` in Fig. 1 is

$$s_6 = s_6' \cup s_6'' \cup s_6''' = \bot \cup s_6'' \cup \bot = s_6''.$$

The abstract state s_6'' shows that the imprecision problem of $\mathcal{E}$, related to the return operation, is still present in $\mathcal{ER}$. By comparing s_2 with s_6'', it can be seen that the return operation makes a very pessimistic assumption about the possible creation points for the `next` and `rotation` fields. In particular, from s_6'' it seems that creation points π_3 and π_4 are reachable (they belong to $\mu^\top$), which is not the case in the concrete state (compare this with σ_6'' in Example 19). As for the domain $\mathcal{E}$, this problem can be solved by including, in the state of the callee, *shadow copies* of the parameters of the caller. This is implemented through a preprocessing of the bodies of the methods which prepend statements of the form $v' := v$ for each parameter v, where v' is the shadow copy of v. Since shadow copies are fresh new variables, not already occurring in the method's body, their value is never changed. In this way, at the end of the method we know which creation points are reachable from the fields of the objects bound to such parameters.

Example 52. Let us reexecute the abstract computation of Example 51, but including shadow copies of the parameters in the abstract states. We denote by p' the shadow copy of the parameter p. We assume that the method `scan` was called with an actual parameter *null* for the formal parameter `n`. The abstract state s_1 contains now two shadow copies

$$s_1 = \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing, \mathtt{n}' \mapsto \varnothing \\ \mathtt{this} \mapsto \{\overline{\pi}\}, \mathtt{this}' \mapsto \{\overline{\pi}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \varnothing \end{bmatrix}.$$

The same change applies to the abstract states s_2 and s_3'' in Example 51. The abstract state s_4'' uses a new shadow copy for the actual parameter of the method `Square.def` i.e.,

$$s_4'' = [\mathtt{this} \mapsto \{\pi_2\}, \mathtt{this}' \mapsto \{\pi_2\}] \star [\mathtt{next} \mapsto \{\pi_2\},\ \mathtt{rotation} \mapsto \varnothing].$$

The static analysis of the method `Square.def` easily concludes that no object has been created. Hence now we have

$$s_5'' = [\mathtt{out} \mapsto *, \mathtt{this}' \mapsto \{\pi_2\}] \star [\mathtt{next} \mapsto \{\pi_2\},\ \mathtt{rotation} \mapsto \{\pi_1\}].$$

Note that the abstract memory is not empty now (compare with Example 51). This is because the shadow variable `this`′ prevents the abstract garbage collector from deleting the creation point π_2 from `next` and the creation point π_1 from `rotation`. Moreover, we know what is reachable at the end of the execution of the `Square.def` method from its parameters (through their shadow copies). Therefore we do not need to apply any pessimistic assumption at return time, and we can define the abstract return operation in such a way that it just transfers the result of the method call into the *res* variable:

$$\mathsf{return}_\tau^\nu(\phi_1 \star \mu_1)(\phi_2 \star \mu_2) = \xi_{\tau[res \mapsto P(\nu)(\mathtt{out})]}(\phi_1[res \mapsto \phi_2(\mathtt{out})] \star \mu_2)$$

so that we have

$$s_6'' = \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing, \mathtt{n}' \mapsto \varnothing \\ \mathtt{this} \mapsto \{\overline{\pi}\}, \mathtt{this}' \mapsto \{\overline{\pi}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \{\pi_1\} \end{bmatrix}.$$

Note that creation points π_3 and π_4 are no longer reachable (compare with Example 51).

As previously noted in Section 1.2, shadow copies of the parameters are also useful for dealing with methods that modify their formal parameters.

There was another problem with $\mathcal{E}$, related to the fact that $\mathcal{E}$ does not distinguish between different variables (see end of Section 4). It is not surprising that $\mathcal{ER}$ solves that problem, as shown below.

Example 53. Let the approximation provided by $\mathcal{ER}$ before the statement `f = new Circle()` in Fig. 1 be

$$s = [\mathtt{f} \mapsto \{\pi_2\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$$

(Example 3). We can compute the approximation *after* that statement by executing two abstract operations. Since the object created at π_3 gets stored inside the variable `f`, we expect the creation point π_2 of the old value of `f` to disappear from the approximation of `f`, which is what actually happens:

$$\begin{aligned} s' &= \mathsf{new}_\tau^{\pi_3}(s) \\ &= \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, res \mapsto \{\pi_3\}, \\ \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\}, \\ \mathtt{rotation} \mapsto \{\pi_1, \pi_4\} \end{bmatrix}, \\ &\mathsf{put\_var}^{\mathtt{f}}_{\tau[res \mapsto \mathtt{Circle}]}(s') \\ &= \xi_\tau \left([\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \{\pi_1, \pi_4\} \end{bmatrix} \right) \\ &= [\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \{\pi_1, \pi_4\} \end{bmatrix}. \end{aligned}$$

Note that the creation points for `rotation` do not disappear, since from the `Circle` bound to `f` it might be possible to reach a `Square` through its `next` field, and a `Square` has a `rotation` field.

5.3 $\mathcal{ER}$ is a refinement of $\mathcal{E}$

We have called $\mathcal{ER}$ a *refinement* of $\mathcal{E}$. In order to give this word a formal justification, we show here that $\mathcal{ER}$ actually includes the elements of $\mathcal{E}$. Namely, we show how every element $e \in \mathcal{E}$ can be *embedded* into an element $\theta(e)$ of $\mathcal{ER}$, such that e and $\theta(e)$ have the same concretisation i.e., they represent the same property of concrete states. The idea, formalised in Definition 54, is that every variable or field must be bound in $\mathcal{ER}$ to all those creation points in e compatible with its type.

Definition 54 (Embedding of $\mathcal{E}$ into $\mathcal{ER}$). Let $s \subseteq \Pi$. We define $\vartheta_\tau(s) \in \mathit{Frame}_\tau^{\mathcal{ER}}$ such that, for every $v \in \mathtt{dom}(\tau)$,

$$\vartheta_\tau(s)(v) = \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{\pi \in s \mid k(\pi) \leq \tau(v)\} & \text{if } \tau(v) \in \mathcal{K}. \end{cases}$$

The *embedding* $\theta_\tau(e) \in \mathcal{ER}_\tau$ of $e \in \mathcal{E}_\tau$ is $\theta_\tau(e) = \xi_\tau(\vartheta_\tau(e) \star \vartheta_{\widetilde{\tau}}(e))$.

Example 55. Let τ_{w_1} be as given in Fig. 6 and $e = \{\overline{\pi}, \pi_1, \pi_2, \pi_3\} \in \mathcal{E}_{\tau_{w_1}}$ (Example 30). Then

$$\theta_{\tau_{w_1}}(e) = \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2, \pi_3\}, \mathtt{n} \mapsto \{\pi_2, \pi_3\} \\ \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix} \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2, \pi_3\} \\ \mathtt{rotation} \mapsto \{\pi_1\} \end{bmatrix}$$

where the missing fields are implicitly bound to $*$ since they have *int* type.

Proposition 56 states that the embedding of Definition 54 is correct. The proof proceeds by showing that $\theta_\tau(e)$ is an element of $\mathcal{ER}_\tau$ and approximates exactly the same concrete states as e, that is, for every element of $\mathcal{E}$ there is an element of $\mathcal{ER}$ which represents exactly the same set of concrete states.

Proposition 56. *Let $\gamma_\tau^{\mathcal{E}}$ and $\gamma_\tau^{\mathcal{ER}}$ be the concretisation maps induced by the abstraction maps of Definitions* 22 *and* 41*, respectively. Then $\gamma_\tau^{\mathcal{E}}(\mathcal{E}_\tau) \subseteq \gamma_\tau^{\mathcal{ER}}(\mathcal{ER}_\tau)$.*

The following example shows that the inclusion relation in Proposition 56 must be strict.

Example 57. Let τ_{w_1} be as given in Fig. 6. By Example 30 we know that $\varnothing \in \mathcal{E}_{\tau_{w_1}}$ and that every $e \in \mathcal{E}_{\tau_{w_1}} \setminus \{\varnothing\}$ must contain $\overline{\pi}$. Moreover, we know that $\{\pi_1\} \notin \mathcal{E}_{\tau_{w_1}}$ and $\{\pi_4\} \notin \mathcal{E}_{\tau_{w_1}}$. Hence

$$\#\mathcal{E}_{\tau_{w_1}} \leq 1 + (\#_\emptyset \wp(\{\pi_1, \pi_2, \pi_3, \pi_4\}) - 2) < 2^4.$$

For what concerns $\mathcal{ER}_{\tau_{w_1}}$, note that for every $e_1, e_2 \in \wp(\{\pi_2, \pi_3\})$ the element

$$[\mathtt{f} \mapsto e_1, \mathtt{n} \mapsto e_2, \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \varnothing, \mathtt{rotation} \mapsto \varnothing, \ldots]$$

is a fixpoint of ξ_τ and hence an element of $\mathcal{ER}_{\tau_{w_1}}$ (Definition 48). Hence

$$\#\mathcal{ER}_{\tau_{w_1}} \geq (\#\wp(\{\pi_2, \pi_3\}))^2 = 2^4 > \#\mathcal{E}_{\tau_{w_1}}.$$

6 Implementation

In this section, we present our practical evaluation of the abstract domain $\mathcal{ER}$. In Section 6.1, we describe the implementation of $\mathcal{ER}$ used to do the experiments and, in Section 6.2, we present the experimental results.

6.1 Analysing Java bytecode

We implemented the abstract domain $\mathcal{ER}$ inside JULIA [44]. This is a generic static analyser written in Java that is designed for analysing full Java bytecode. Generic means that JULIA does not embed any abstract domain but, instead, can be instantiated for a specific static analysis once an appropriate abstract domain and the attached abstract operations are provided. For instance, JULIA can perform *rapid type analysis* (a kind of *class analysis* [4]) or instead escape analysis through $\mathcal{ER}$ by simply swapping these abstract domains.

In order to target the escape analysis of real Java bytecode programs, the implementation had to address a number of problems due to features of the Java bytecode itself. We describe the main problems and how we addressed them. These problems were:

1. the Java Virtual Machine frame contains both local variables and an operand stack and the number of elements in the operand stack can change within the same method, although its size at a given program point is fixed and statically known;
2. a very large number of library classes are likely to be called and, hence, would need to be analysed;
3. Java bytecode is unstructured i.e., lacking any explicit scope structure and code is weaved through an extensive use of explicit `goto` jumps;
4. since the Java bytecode makes extensive use of exceptions, the control flow for exceptions must also be considered.
5. Java bytecode has static fields, which are like global variables of traditional imperative languages, and are always in scope, so that the objects bound to static fields cannot be garbage-collected.

We solved Problem (1) by rewording our notion of frame (Definition 35) into a set of local variables and a stack of variables. The number of stack variables (elements) in a given program point is statically determined since `.class` files must be verifiable [33]. Since Java bytecode holds intermediate results in the operand stack, the latter plays the role of our *res* variable.

We dealt with Problem (2) by analysing some library classes only, and making *worst-case assumptions* [16] about the behaviour of calls to methods of other classes. This means that we assume that such calls can potentially do everything, such as storing the parameters into (instance or class) fields or returning objects created in every creation point π (with the restriction, however, that π creates objects of class compatible with the return type of the method). It is easy to see how an extensive use of this policy quickly leads to imprecision. The situation is made worse in Java (bytecode) because of constructor chaining, stating that every call to a constructor eventually leads to a constructor in the *library* class `java.lang.Object` [3]. To cope with these problems, we allowed the analyser to

access at least the code of `java.lang.Object`. We also allowed the analyser access to yet more library classes, leading to more precise but also more costly analyses. For many native methods, whose Java bytecode is not available, we have provided hand-made bytecode stubs which agree with the declared abstract behaviour of the methods.

Problem (3) was solved by building a graph of blocks of codes, each bound to all its possible successors in the control-flow. We use class hierarchy analysis to deal with virtual calls whose target is not explicitly embedded in the code [18]. Java bytecode subroutines (i.e., the `jsr`/`ret` mechanism) are handled by linking each block ending with a `jsr` to the block starting with its target. The block ending with `ret` is then conservatively linked with all blocks starting with an instruction immediately following a `jsr` bytecode *in the same method* of `ret`. The restriction to the same method is correct because of a constraint imposed on valid Java bytecode by the verification algorithm [33]. The resulting graph is the same as that of *dominators* that are defined in [2] for much simpler languages. The graph is then used for a fixpoint computation by following the structure of its strongly connected components.

We solved Problem (4) by using the technique pioneered in [30]. It consists in denoting a piece of code c through a map from the input state to the *normal* output state *and* an *exceptional* output state, representing the state of the Java virtual machine if an exception has been thrown inside c. Composition of commands uses the normal output state [45], but composition with exception handlers uses the exceptional final state.

We dealt with Problem (5) by modifying the abstract garbage collector of Definition 44 so that it does not garbage collect the creation points reachable from static fields. Technically, this amounts to adding to the map ρ_τ of Definition 44 the creation points bound to the approximation of the static fields of the classes of the program.

The abstract domain $\mathcal{ER}$ is implemented inside JULIA as a Java class extending an abstract (in the sense of Java [3]) class standing for a generic abstract domain. This class contains methods that compute the denotation of every single Java bytecode (i.e., denotations similar to those given in Fig. 10 for our simplified bytecode).

The choice of representation for a denotation affects the speed of the analysis. The representation we chose was a set-constraint [20] between its input and output variables. For instance, the denotation for the new^π operation in Fig. 10 is implemented as a constraint $\{\pi\} \subseteq S$ over the unknown S. It states that the creation point π must belong to the set S of the creation points for *res* in the output of the operations (i.e., for the top of the operand stack when considering the real Java bytecode). We use $\subseteq$ instead of $=$ since there might be many possible ways of reaching the program point that follows new^π. We use default reasoning to state that the other variables are unchanged. This is a generalisation of the technique we introduced in [28]. As another example, the denotation for the $\mathsf{get\_var}^v$ operation in Fig. 10 is implemented as a constraint $S_1 \subseteq S_2$ over the unknowns S_1 and S_2. It states that the set S_2 of creation points for *res* in the output must contain the set S_1 of creation points for v in the input. We solve the set-constraints constructed from a program by propagation of creation points. Namely, a constraint such as $\{\pi\} \subseteq S$ propagates π into S. A constraint such as $S_1 \subseteq S_2$ propagates the creation points inside S_1 into S_2. Propagation starts with empty approximations for the unknowns and continues until there is no further growth in these approximations. We have implemented this propagation by exploiting a preliminary topological sort of the unknowns of the constraints. Namely, a constraint $S_1 \subseteq S_2$ induces a pre-order (a reflexive and transitive relation) $S_1 \leq S_2$. A topological sort w.r.t. this pre-order builds a tree of strongly-connected components of unknowns. A strongly-connected component represents a set of mutually dependent unknowns. We propagate the creation points by following the topological ordering backwards, so that we can consider one strongly-connected component at a time. This technique significantly speeds up the propagation.

There were two alternative choices we might have taken for the representation. The simplest would have been an extensional definition, in the form of an exhaustive input/output tabling; but that would have been far too slow. Alternatively, we could have used binary decision diagrams [9] to represent the denotations; this traditional approach can represent the denotations in a compact and efficient way. This technique is certainly possible, but it requires more technical work since we have to code maps over sets of creation points through Boolean functions. Moreover, since bytecodes usually apply local modifications to the state (for instance, the put_var^v bytecode in Fig. 10 leaves all variables other than v untouched), they would be coded into binary decision diagrams which mainly assert that the output variables are a copy of their input counterparts. In terms of Boolean functions, this means that such functions would contain a lot of *if and only if* constraints, which significantly increases the size of the diagram. By using set-constraints, we solve this problem through default reasoning.

The use of set-constraints is appealing since we can easily use the same unknown to represent two or more distinct approximations. For instance, different unknowns might represent the approximation of a field at different program points, or rather the same unknown might represent all those approximations. The second choice leads to a less precise analysis but also to fewer unknowns and constraints than the first choice. Thus, the second choice should lead to faster analyses. In the first case we say that the approximation of the fields is *flow-sensitive*, while in the second case we say that it is *flow-insensitive*. The same idea can be used for the approximations of local variables or even operand stack elements. In Section 6.2 we evaluate the practical consequences of merging different approximations into the same unknown.

The use of set-constraints for the representation has, however, also a few negative consequences. To keep the implementation simple and fast, our set-constraints are built from equality, union and intersection only. But these operations do not allow us to represent tests on the input variables (such as in put_field, see Fig. 10). Hence conservative approximations must be made. For instance, the third case of the definition of put_field is *always* used. This is correct since it is a conservative approximation of all three cases. Moreover, it does not introduce a significant precision loss since the first alternative of the definition of put_field deals with the pathological case when a given put_field is *always* applied to a *null* receiver; and the second alternative deals with the case when a given put_field is *always* applied to a receiver which is made unreachable by the evaluation of the value which must be put inside the field. Both the first two alternatives correspond to legal but quite unusual ways of using put_field and are almost never applicable. For efficiency reasons, the ξ garbage collector is only used at the end of a method. This is a correct approximation since ξ is an lco (Proposition 46) although *forgetting* some of its applications can lose precision. Also for efficiency reasons, we have used memoisation to cache repeated calls to the abstract garbage collector.

We have described how we map the input abstract state to the output (and exceptional) abstract state. However, the information needed for stack allocation is related to some internal points of the program. For instance, in the program in Fig. 1, we would like to know if the creation point π_4 inside `rotate` could be stack allocated. For this, we need to know the set of the creation points of objects that are reachable at the end of `rotate` from each return value, from the (possible) objects thrown as an exception, or from the fields of the objects bound to its parameter (i.e., the set E of Section 1.2). Therefore, we need information related to some internal program points. This can be obtained by placing a *watchpoint* at every exit point of a method such as `rotate`. Note that, with the analyser JULIA, we can do this automatically and obtain the set of creation points that can be stack allocated.

6.2 Experimental evaluation

We report our experiments with the escape analysis through $\mathcal{ER}$ of some Java applications: `Figures` is the program in Fig. 1 fed with a list of `Circles`; `LimVect` is a small Java program used in [6]; `Dhrystone` version 2.1 is a testbench for numerical computations (most of the arrays it creates can be stack allocated); `ImageVwr` is an image visualisation applet; `Morph` is an image morphing program; `JLex` version 1.2.6 is the Java version of the well-known `lex` lexical analysers generator; `JavaCup` version 0.10*j* and `Javacc` version 3.2 are compilers' compilers; `Julia` version 0.39 is our JULIA analyser itself; `Jess` version 6.1*p*7 is a rule engine and scripting language for developing rule-based expert systems. All these benchmarks are free software except `Jess` which is copyrighted. Some of these programs were analysed in [6]; these are `LimVect`, `Dhrystone`, `JLex`, an older version of `Javacc` and an older version of `Jess`. Note that the newer versions of `Javacc` and `Jess` considered here are bigger than those used in [6].

Our experiments have been performed on a Pentium 2.1 Ghz machine with 1024 megabytes of RAM, running Linux 2.6 and Sun Java Development Kit version 1.5.0 with HotSpot just-in-time compiler.

For each experiment, we report how many Java classes, methods and bytecodes are analysed, as well as the time taken by the analysis (in seconds) to build and solve the set of constraints generated for our escape analysis. We first show the *static* precision of the analyses (Section 6.2.1). Namely, we report the number of creation points which can be stack allocated. We study how the precision of the analyses is affected by flow sensitivity and by the ability to approximate precisely each field. Later (Section 6.2.2), we report the *dynamic* precision of the analyses i.e., the number of creation operations which are stack allocated *at run-time* and their relative ratio w.r.t. those which are heap allocated. We also provide information on the amount of memory which is stack allocated rather than heap allocated at run-time. Finally (Section 6.2.3) we briefly discuss the cost of the analyses.

6.2.1 Static tests

We start from the fastest but also less precise way of using our escape analysis. Namely, the analyses are completely flow insensitive and field insensitive, in the sense that the fields are approximated into one variable and, except for `java.lang.Object`, library classes are not included. As we said in Section 6.1, calls to methods of other library classes are approximated through a worst-case assumption. In particular, this assumption states that the parameters passed to the call escape, since they might be stored into a static field, and hence be accessible after the call has returned. Because of constructor chaining, all object creations result in a call to the constructor of `java.lang.Object`. This is why the inclusion of that class is a minimum requirement to the precision of the analysis. Otherwise, every newly created object would escape as soon as it is initialised. The results are shown in Fig. 11, where for each benchmark we report the number of classes, methods and bytecodes analysed and the time of the analysis (in seconds).

Figure 11 also reports the number of set-constraints generated for the analysis. These constraints are organised into a graph. Each variable in the constraints is a node in the graph; nodes are connected if they are related by some constraint. The *linearity* column reports the average size of a strongly-connected component. Linearity is equal to 1.000 for fully non-recursive programs without cycles. Higher values of linearity represent programs which use recursion and cycles extensively. For a given number of constraints, their solution is computed more efficiently if linearity is low.

benchmark	clss	meth	bytec	time	*SA*		*TT*	*NC*	*LIN*
`Figures`	6	17	146	0.06	1	(20%)	0.41	109	2.067
`LimVect`	2	8	48	0.02	1	(0%)	0.42	46	1.631
`Dhrystone`	7	24	610	0.17	4	(25%)	0.28	253	1.640
`ImageVwr`	3	23	1238	0.35	0	(0%)	0.28	412	1.669
`Morph`	1	14	1367	0.30	0	(0%)	0.22	253	1.355
`JLex`	26	138	12520	0.76	2	(0%)	0.06	2904	1.974
`JavaCup`	37	317	14390	1.76	1	(0%)	0.12	5577	2.206
`Julia`	164	821	28507	10.45	6	(0%)	0.37	17653	2.672
`Jess`	268	1543	51663	31.04	2	(0%)	0.60	45614	2.914
`Javacc`	65	953	79325	15.32	0	(0%)	0.19	17759	2.162

Fig. 11 Flow insensitive escape analyses with $\mathcal{ER}$. Fields are merged. Only `java.lang.Object` is included in the analysis. *SA* is the number of creation points which are stack allocated; *TT* is the time per one thousand bytecodes; *NC* is the number of constraints generated; *LIN* is the linearity of the set of constraints

benchmark	clss	meth	bytec	time	*SA*		*TT*	*NC*	*LIN*
`Figures`	6	17	146	0.08	3	(60%)	0.54	454	1.003
`LimVect`	2	8	48	0.06	1	(33%)	1.25	214	1.007
`Dhrystone`	7	24	610	0.20	8	(50%)	0.33	2784	1.174
`ImageVwr`	3	23	1238	0.50	0	(0%)	0.40	7831	1.377
`Morph`	1	14	1367	0.35	0	(0%)	0.26	6483	1.110
`JLex`	26	138	12520	2.40	11	(5%)	0.19	60379	1.298
`JavaCup`	37	317	14390	14.98	15	(3%)	1.04	124097	1.578
`Julia`	164	821	28472	33.30	30	(3%)	1.17	217874	1.788
`Jess`	268	1543	51663	51.77	51	(3%)	1.00	335010	1.645
`Javacc`	65	953	79325	239.22	45	(3%)	3.01	382862	1.629

Fig. 12 Flow sensitive (on the operand stack only) escape analyses with $\mathcal{ER}$. Fields are merged. Only `java.lang.Object` is included in the analysis

Although the analyses in Fig. 11 are relatively fast, it can be seen that almost no creation points are found to be stack allocatable. The analyses can be made more precise if flow sensitivity is used, at least for the operand stack. Results using this level of flow sensitivity are shown in Fig. 12. They are more precise than those in Fig. 11, but the analyses are also more expensive. If we also analyse some library classes, the precision of the analyses improves further. Namely, we decided to add part of the `java.lang` and `java.util` standard Java packages. Such classes are chosen in such a way to include typical candidates for stack allocation and to form an upward closed set, so that constructor chaining for those classes never goes out of the set of analysed classes. The results with these additions are shown in Fig. 13. We only count the creation points inside the classes of the application, so that numbers are comparable with those in Figs. 11 and 12. In comparison with Fig. 12, we manage to stack allocate many more creation points, but with a further increase in the cost of the analyses.

The final experiments used the full power of the $\mathcal{ER}$ abstract domain by providing a (flow insensitive) specific approximation for each field. The results are shown in Fig. 14. The precision is just slightly better than in Fig. 13, and the analyses require less time. They are

benchmark	clss	meth	bytec	time	SA		TT	NC	LIN
Figures	6	17	146	0.10	3	(60%)	0.68	454	1.003
LimVect	4	11	1057	0.18	1	(33%)	0.17	2743	1.007
Dhrystone	14	54	2240	0.35	12	(75%)	0.16	7157	1.116
ImageVwr	47	209	8557	1.64	1	(8%)	0.19	43152	1.242
Morph	18	93	7302	1.20	1	(5%)	0.16	33512	1.168
JLex	72	396	21025	3.71	48	(22%)	0.18	94243	1.248
JavaCup	84	569	23196	11.98	213	(39%)	0.52	147139	1.434
Julia	530	2007	60846	50.47	331	(41%)	0.83	348621	1.551
Jess	363	2151	71329	53.71	289	(22%)	0.75	399188	1.539
Javacc	160	1489	97981	65.75	878	(61%)	0.67	418552	1.436

Fig. 13 Flow sensitive (on the operand stack only) escape analyses with the abstract domain $\mathcal{ER}$. Fields are merged. The standard library classes `java.lang.{Object, CharSequence, String*, AbstractStringBuilder, Integer, Number, Character}` and `java.util.{AbstractList, AbstractCollection, Vector, HashMap, Hashtable, AbstractMap}` are included in the analysis. For `Julia`, we also included the bcel libraries for bytecode manipulation

benchmark	clss	meth	bytec	time	SA		TT	NC	LIN
Figures	6	17	146	0.13	3	(60%)	0.89	454	1.003
LimVect	4	11	1057	0.18	1	(33%)	0.17	2743	1.001
Dhrystone	14	54	2240	0.34	12	(75%)	0.15	7157	1.080
ImageVwr	47	209	8557	1.76	1	(8%)	0.20	43174	1.218
Morph	18	93	7302	1.14	1	(5%)	0.16	33526	1.158
JLex	72	396	21025	3.11	49	(23%)	0.15	94383	1.153
JavaCup	84	569	23196	9.24	215	(39%)	0.40	147159	1.319
Julia	530	2007	60846	38.61	336	(41%)	0.63	348799	1.428
Jess	363	2151	71329	56.71	289	(22%)	0.79	399289	1.488
Javacc	160	1489	97981	59.45	895	(62%)	0.61	419067	1.378

Fig. 14 Flow sensitive (on the operand stack only) escape analyses with the abstract domain $\mathcal{ER}$. A specific approximation is used for each field. The same library classes as in Fig. 13 are included in the analysis. For `Julia`, we also included the bcel libraries for bytecode manipulation

sometimes even faster than those in Fig. 12. This reduction in time might seem surprising. However, this is a consequence of the fact that a field-specific approximation slightly increases the number of constraints, but reduces linearity and the average size of creation point sets; hence, less time is needed to solve the constraints (compare the linearity columns in Figs. 13 and 14). A similar behaviour has been experienced in [37]. More generally, it has been witnessed in different contexts that increasing the precision of a static analysis may yield faster computations, since an imprecise analysis yields spurious execution paths which slow down the analysis itself.

Beyond these experiments, we also tried to include more library classes in the analysis (such as all `java.lang` and `java.util` packages), but the results were very similar to those in Fig. 14, confirming the claim in [6] that most of the stack allocatable objects are arrays, `java.lang.StringBuffers` and a few objects of the collection classes (vectors and sets). We also tried to use flow sensitivity for the local variables and the field approximations but this did not improve the results. This behaviour can be explained, for local variables,

by observing that typical Java compilers do not try to recycle local variables if they can be used for different tasks in different parts of a method. Hence one approximation per method is enough. Note that both conclusions agree with the results provided in [12] where flow sensitivity looks useless for stack allocation and a bounded field approximation reduces the precision of stack allocation in at most one case in ten. The analysis in [12] works for Java instead of Java bytecode, so flow sensitivity for the operand stack is meaningless in their case.

The precision of the analyses seems similar to that of the experiments reported in [6]. The results reported in Fig. 14 for the first five benchmarks are actually optimal, in the sense that no other creation point can ever be stack allocated. For the other five benchmarks, the exact comparison is hard since the older versions of the benchmarks, as analysed in [6], are not available anymore. See, however, Section 7 for a theoretical comparison.

The overall conclusion we draw from our experiments is that flow sensitivity is important, but only for the stack variables. The inclusion of library classes is also essential for the precision although, in practice, only very few classes are needed. The ability to approximate each field individually does not contribute significantly to the precision of the analyses, but improves their efficiency.

6.2.2 Dynamic tests

The static measurements in Section 6.2.1 have been useful to compare the relative precision and cost of different implementations of our escape analysis. However, another piece of information, important for an escape analysis, is the number of creation operations that are actually avoided *at run-time* because their creation point has been stack allocated. As well as the size of the objects which are stack allocated w.r.t. the size of the objects which are heap allocated. We computed these measurements for the analyses reported in Fig. 14 only, which are the most precise escape analyses which we managed to implement with $\mathcal{ER}$. Some results are shown in Fig. 15. For each benchmark, we report the number of objects and the amount of memory allocated in the stack or in the heap. In Section 7, these results are compared with results reported for other escape analysers. Here we just note that the poor result for the escape analysis of `Julia` is a consequence of the fact that `Julia` mainly computes a large set of constraints which escape from their creating methods to flow into the methods that solve them. We note that escape analysis is of little use for this type of program.

6.2.3 Cost of the analysis

Figure 14 shows that one minute is more than enough to analyse each of the benchmarks, some of them featuring more than 2000 methods. The JULIA analyser is under development, so that analysis times can only improve. In theory, as a consequence of using sets of creation points as variable approximations (Section 6.1), their computation as a fixpoint of a system of set constraints might require an exponential number of iterations. Thus, an important observation from the same Fig. 14 is that there is no exponential blow-up of the analysis time with the size of the benchmarks (see the *time per* 1000 *bytecodes* column *TT*); hence, it appears from these results that the worst-case scenario, leading to an exponential blow-up, is not frequent in practice. Moreover, from Fig. 14 it seems that the cost of the analysis is not only related to the size of the benchmark, but also to its linearity. See for instance the case of `JLex` and `JavaCup` in Fig. 14, which have comparable size (i.e., number of bytecodes) but quite different linearity.

benchmark	objects in the stack		objects in the heap	
Figures	101	97.11%	3	2.89%
LimVect	1	33.33%	2	66.66%
Dhrystone	200009	99.99%	4	0.01%
JLex	672	8.92%	6854	91.08%
JavaCup	141875	62.37%	85564	37.63%
Julia	5534	6.56%	78748	93.44%
Jess	924	7.81%	10897	92.19%
Javacc	26461	43.72%	34050	56.28%

benchmark	memory in the stack		memory in the heap	
Figures	2024	98.82%	24	1.18%
LimVect	12	30.00%	28	70.00%
Dhrystone	1600140	96.03%	66144	3.97%
JLex	147060	47.77%	160772	52.23%
JavaCup	1580336	48.76%	1660516	51.24%
Julia	70764	5.57%	1198120	94.47%
Jess	13328	4.90%	258128	95.10%
Javacc	604244	40.72%	883224	59.28%

Fig. 15 The dynamic statistics for our benchmarks. Memory is expressed in bytes. `Dhrystone` performs its numerical benchmark 100000 times; `JLex` is applied to `sample.tex`. `JavaCup` is applied to the grammar `tiger.cup` (included in the distribution of JULIA) with the `-dump_states` option on. `Julia` is applied to the escape analysis of `Dhrystone` performed as in Fig. 14. `Javacc` is applied to the grammar `Java1.1.jj`. `Jess` is applied to the solution of `fullmab.clp`

7 Discussion

The first escape analysis [36] was designed for functional languages where lists are the representative data structure. Hence the analysis was meant to stack allocate portions of lists which do not escape from the function which creates them. That analysis was later made more efficient in [19] and finally extended to some imperative constructs and applied to very large programs [5].

More recently, escape analysis has been studied for object-oriented languages. In this context, there are actually different formalisations of the meaning of *escaping*. While we assume that an object o escapes from its creating method if it is still reachable after this method terminates, others (such as [38, 41, 50]) further require that o is actually used after the method terminates. This results in less conservative and hence potentially more precise analyses than ours. Note, however, that our notion of *escaping* allows us to analyse libraries whose calling contexts are not known at the time of the analysis.

The first work which can be related to escape analysis seems to us to be [39] where a *lifetime analysis* propagates the *sources* of data structures. This same idea has been used in [1] where the traditional reaching definition analysis is extended to object-oriented programs. Note that, to improve efficiency, [1] made the escape analysis demand-driven.

Many escape analyses use however some graph-based representation of the memory of the system, typically derived from previous work on points-to analysis. Escaping objects are then identified by applying some form of reachability on this graph. Examples are [12, 41, 49–51]. Points-to information lets such analyses select the target of virtual calls on the basis of the class of the objects pointed to by the receiver of the virtual call. Although these works abstract *the memory graph* into a graph while we abstract *the state* into $\mathcal{E}$ or $\mathcal{ER}$, we think that the information expressed by $\mathcal{E}$ or $\mathcal{ER}$ can be derived by abstracting such graphs. Hence our analyses should be less precise but more efficient than [12, 41, 49–51]. Namely, we miss

the sharing information contained in a graph-based representation of the memory of the system. Note that, to improve efficiency, some escape analyses such as [49] have been made incremental.

A large group of escape analyses use constraints (typically, set-constraints) for expressing escape analyses. These include [7, 21, 37, 38, 46], although [46] assumes that points-to information is available at the time of the analysis. The escape analysis in [6] is unique in that it uses integer *contexts* to specify the part of a data structure which can escape. However, integer contexts are an approximation of the full typing information used in our abstract domain $\mathcal{ER}$ as well as in most of the previously cited escape analyses. This possible imprecision has been observed also by [12] where it is said (without formal proof) that their analysis is *inherently more precise* than that defined in [6]. The simplicity of Blanchet's escape analysis is however appealing, and the experimental times he reports in [6] currently score as the fastest of all the cited analyses that provide timings.

The way we deal with exceptions (Section 6.1) is largely inspired by [30]. It is also similar to the technique in [10], although their optimised factorisation is not currently implemented inside our JULIA analyser. Once implemented, we expect it to improve the overall efficiency of the analyses shown in Fig. 14.

Some escape analyses have been formally proved correct. Namely, [51] has been proved correct in [40], and [12] in [11]. Neither proof is based on abstract interpretation, and no optimality result is proved. The proof in [6] is closer to ours, since it is based on abstract interpretation. However, a Galois connection is proved to exist between the concrete and the abstract domain, rather than a Galois insertion. Moreover, no optimality result for the abstract operations is provided.

To the best of our knowledge, our notion of abstract garbage collector (Definitions 25 and 44) and its use for deriving Galois *insertions* rather than *connections* (Propositions 31 and 49) is new. A similar idea has been used in [12], which removes from the connection graphs (their abstraction) that part that consists only of captured nodes (unreachable from parameters, result and static variables). However, this was not related to the Galois insertion property. Static types of variables are used in [25] to improve the precision of a points-to analysis by removing spurious points-to sets which are not allowed by the static typing of the variables. This idea is somehow similar to our garbage-collectors, but no connection is shown to the Galois insertion property.

It is quite hard to compare the available escape analyses w.r.t. precision. From a theoretical point of view, their definitions are sometimes so different that a formal comparison inside the same framework is impossible. However, below we make some informal comparisons and discuss some of the issues that affect the precision.

- Escape analyses using graph-based representations of the heap should be the most precise overall since they express points-to and sharing information [12, 41, 49–51].
- Precision is also significantly affected by the call-graph construction algorithm used for the analysis which is largely independent from the analysis itself [24, 48]. As $\mathcal{ER}$ couples each variable with the set of its creation points, class information can be recovered and the call-graph constructed. This is also typical of those analyses that compute some form of points-to information.
- Another issue related to precision is the level of flow and context sensitivity of the analysis. Our experiments (Section 6.2) have shown that flow sensitivity is important for the operand stack only. This agrees with the experiments reported in [12], since the operand stack is a feature of Java bytecode not present in Java (the target language of [12]). Context sensitivity i.e., the ability to name a given creation point in method *m* with different names, depending

on the call site of m, is advocated as an important feature of escape analyses [6, 49, 51]. This idea is taken further in [50], where method bodies are *cloned* for each different calling context in order to improve the precision of the analysis. Our analysis decorates each given program point with a unique name, and hence misses this extra axis of precision. Note, however, that method cloning can safely be applied before many escape analyses, including ours.

– The optimality of the abstract operations or algorithm used for the analysis also affects its precision, since optimality entails that the analysis uses the abstract information in the most precise possible way. To the best of our knowledge, we are the first to prove an optimality result for the abstract operations of an escape analysis, which is a significant step forward although we are aware that the composition of optimal operations to build an abstract semantics is not necessarily optimal.
– A final source of precision comes from the preliminary inlining of small methods before the actual analysis is applied. Inlining can only enlarge the possibilities for stack allocation, although care must be taken to avoid any exponential code explosion. As far as we can see, only [6] applies method inlining before the escape analysis. This technique is largely independent from the subsequent escape analysis, so it could be applied with other escape analyses, including our own.
– Some specially designed analyses should perform better on some specific applications. For instance, the analysis in [38] is probably the most precise one w.r.t. synchronisation elimination. This is because it allows one to remove unnecessary synchronisation (locking) on objects which do escape from their creating thread provided that, at run-time, they are locked by at most one thread. As another example the analysis in [41] is expected to perform better on multithreaded applications, since it models precisely inter-thread interactions. All other escape analyses (including ours) conservatively assume instead that everything stored inside a thread object (and the thread object itself) escapes.

From an experimental point of view, a comparison of different escape analyses is possible although hard and sometimes contradictory. This difficulty is due to the fact that some analyses have been evaluated w.r.t. stack allocation, others w.r.t. synchonization elimination, and others w.r.t. both. Moreover, sometimes only compile-time (*static*) statistics are provided (such as [49]), sometimes only run-time (*dynamic*) statistics (such as [6]), sometimes both (such as [12] and ourselves). Still, some statistics include the library classes (such as [6, 7, 38]), others only report numbers for the user classes (such as [12], which analyses library code during the analysis but does not transform it for stack allocation, exactly as we do in Section 6.2). Furthermore, there is no standard set of benchmarks evaluated across all different escape analyses. If some benchmark is shared by different analyses, their version number is not necessarily the same. For what concerns the dynamic statistics, the input provided to the benchmark is important. Hence we provided this information in Fig. 15, trying to make it as similar to that in [6] as possible. However, others do not specify this information (such as [12]). Finally, analysis times are not always disclosed, making a fair comparison harder. Let us anyway compare the benchmarks that we share with [6], [12] and [37]. From Fig. 15, we see that we perform equally on `Dhrystone` w.r.t. [6] (which, however, does not specify the parameter passed to `Dhrystone`, which completely modifies the run-time behaviour of `Dhrystone`). For `JLex`, we stack allocate 47.77% of the memory while [6] stack allocates 26%; we stack allocate 8.92% of the run-time objects, while [37] stack allocates 31.6%; we found 23% of the allocation sites to be stack allocatable (Fig. 14) while [37] found 27%. For `JavaCup`, we stack allocate 48.76% of the memory, while [12] stack allocates 17%. We stack allocate 39% of the allocation sites, while [37] stack allocates 30%. For `Jess`, we

only stack allocate 4.90% of the memory, while [6] manages to stack allocate 27% and [37] 17.9%. This case may be a consequence of a preliminary lack of methods inlining, since the `Jess` program actually contains a large set of very small methods. For `Javacc`, we stack allocate 40.72% of the memory, while [6] stack allocates 43%; we stack allocate 43.72% of the run-time objects, while [37] stack allocates 45.8%; this is contradictory with the fact that we found 62% of the allocation sites to be stack allocatable (Fig. 14) while [37] found only 29%.

8 Conclusion

We have presented a formal development of an escape analysis by abstract interpretation, providing optimality results in the form of a Galois insertion from the concrete to the abstract domain and of the definition of optimal abstract operations. This escape analysis has been implemented and applied to full Java (bytecode). This results in an escape analyser which is probably less precise than others already developed, but still performs well in practice from the points of view of its cost and precision (Sections 6.2 and 7).

A first, basic escape domain $\mathcal{E}$ is defined as a property of concrete states (Definition 29). This domain is simple but non-trivial since

- The set of the creation points of the objects reachable from the current state can both grow (`new`) and shrink (δ); i.e., *static type information contains escape information* (Examples 24 and 33);
- That set is useful, sometimes, to restrict the possible targets of a virtual call i.e., *escape information contains class information* (Example 33).

However, the escape analysis induced by our domain $\mathcal{E}$ is not precise enough from a computational point of view, since it induces rather imprecise abstract operations. We have therefore defined a refinement $\mathcal{ER}$ of $\mathcal{E}$, on the basis of the information that $\mathcal{E}$ lacks, in order to attain better precision. The relation between $\mathcal{ER}$ and $\mathcal{E}$ is similar to that between Palsberg and Schwartzbach's class analysis [35, 45] and *rapid type analysis* [4] although, while all objects stored in memory are considered in [4, 35, 45], only those actually reachable from the variables in scope are considered by the domains $\mathcal{E}$ and $\mathcal{ER}$ (Definitions 22 and 41). The ability to describe only the reachable objects, through the use of an abstract garbage collector (δ in Fig. 9 and ξ in Fig. 10), improves the precision of the analysis, since it becomes focused on only those objects that can actually affect the concrete execution of the program.

It is interesting to consider if this notion of reachability and the use of an abstract garbage collector can be applied to other static analyses of the run-time heap as well. Namely, class, shape, sharing and cyclicity analyses might benefit from them.

Acknowledgments This work has been funded by the Italian MURST grant *Abstract Interpretation, Type Systems and Control-Flow Analysis* and by the British EPSRC grant GR/R53401.

References

1. Agrawal, G.: Simultaneous demand-driven data-flow and call graph analysis. In: Proc. of the International Conference on Software Maintenance (ICSM'99), pp. 453–462. IEEE Computer Society, Oxford, UK (1999).
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles Techniques and Tools. Addison Wesley Publishing Company (1986).

3. Arnold, K., Gosling, J., Holmes, D.: The Java™ Programming Language. 3rd edn, Addison-Wesley (2000).
4. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proc. of OOPSLA'96, vol. 31(10) of ACM SIGPLAN Notices, pp. 324–341. ACM Press, New York (1996).
5. Blanchet, B.: Escape analysis: correctness proof, implementation and experimental results. In: 25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages (POPL'98), pp. 25–37. ACM Press, San Diego, CA, USA, (1998).
6. Blanchet, B.: Escape analysis for java: theory and practice. ACM TOPLAS, **25**(6), 713–775 (2003).
7. Bogda, J., Hölzle, U.: Removing unnecessary synchronization in java. In: Proc. of OOPSLA'99, vol. 34(10) of SIGPLAN Notices, pp. 35–46. Denver, Colorado, USA (1999).
8. Bossi, A., Gabbrielli, M., Levi, G., Martelli, M.: The s-semantics approach: theory and applications. J. Logic. Program. **19/20**:149–197, 1994.
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans Comput. **35**(8), 677–691 (1986).
10. Choi, J.-D., Grove, D., Hind, M., Sarkar, V.: Efficient and precise modeling of exceptions for the analysis of java programs. In: Proc. of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'99), pp. 21–31. Toulouse, France (1999).
11. Choi, J.-D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for java using escape analysis. Technical Report RC22340, IBM (2002).
12. Choi, J.-D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for java using escape analysiss. ACM TOPLAS. **25**(6), 876–910 (2003).
13. Cortesi, A., Filé, G., Winsborough, W.: The quotient of an abstract interpretation. Theoret. Comput. Sci., **202**(1–2), 163–192 (1998).
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77, pp. 238–252 (1977).
15. Cousot, P., Cousot, R.: Abstract interpretation and applications to logic programs. J Logic Program, **13**(2 & 3):103–179 (1992).
16. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.), Proceedings of Compiler Construction, vol. 2304 of Lecture Notes in Computer Science, pp. 159–178. Springer-Verlag, Grenoble, France (2002).
17. Dams, D.R.: Abstract interpretation and partition refinement for model checking. PhD thesis, Eindhoven University of Technology, The Netherlands (1996).
18. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W.G. (ed.), Proc. of ECOOP'95, vol. 952 of *LNCS*, pp. 77–101. Springer-Verlag, Århus, Denmark, (1995).
19. Deutsch, A.: On the complexity of escape analysis. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), pp. 358–371. ACM Press, Paris, France (1997).
20. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans Program Lang Sys. **22**(5), 861–931 (2000).
21. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: Watt, D.A. (ed.), Compiler Construction, 9th International Conference (CC'00), vol. 1781 of *Lecture Notes in Computer Science*, pp. 82–93. Springer-Verlag, Berlin (2000).
22. Giacobazzi, R., Ranzato, F.: Refining and compressing abstract domains. In: Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97), vol. 1256 of *LNCS*, pp. 771–781. Springer-Verlag (1997).
23. Giacobazzi, R., Scozzari, F.: A logical model for relational abstract domains. ACM Trans Program Lang Sys. **20**(5), 1067–1109 (1998).
24. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM TOPLAS. **23**(6), 685–746 (2001).
25. Lhoták, H., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.), Proc. of Compiler Construction, vol. 2622 of *Lecture Notes in Computer Science*, pp. 153–169. Springer-Verlag, Warsaw, Poland (2003).
26. Hill, P.M., Spoto, F.: A foundation of escape analysis. In: Kirchner, H., Ringeissen, C. (ed.), Proc. of AMAST'02, vol. 2422 of *LNCS*, pp. 380–395. Springer-Verlag, St. Gilles les Bains, La Réunion island, France (2002).
27. Hill, P.M., Spoto, F.: A refinement of the escape property. In: Cortesi, A. (ed.), Proc. of the VMCAI'02 workshop on Verification, Model-Checking and Abstract Interpretation, vol. 2294 of *Lecture Notes in Computer Science*, pp. 154–166. Springer-Verlag, Venice, Italy (2002).
28. Hill, P.M., Spoto, F.: Logic programs as compact denotations. Comput. Lang. Sys. Struct. **29**(3), 45–73 (2003).

29. Hill, P.M., Spoto, F.: Deriving escape analysis by abstract interpretation: proofs of results. The Computing Research Repository (CoRR): arXiv:archive.cs/PL/0607101 (2006).
30. Jacobs, B., Poll, E.: Coalgebras and monads in the semantics of Java. Theoret. Comput. Sci. **291**(3), 329–349 (2003).
31. Jensen, T.: Disjunctive program analysis for algebraic data types. ACM Trans. Program. Lang. Syst. **19**(5), 752–804 (1997).
32. Jones, N.D., Søndergaard, H.: A semantics-based framework for the abstract interpretation of prolog. In: Abramsky, S., Hankin, C. (eds.), Abstract Interpretation of Declarative Languages, pp. 123–142. Ellis Horwood Ltd (1987).
33. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification. 2nd edn, Addison-Wesley. (1999).
34. Cousot. P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL'79), pp. 269–282. ACM, San Antonio, Texas (1979).
35. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pp. 146–161. ACM Press (1991).
36. Park, Y.G., Goldberg, B.: Escape analysis on lists. In: ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92), vol. 27(7) of SIGPLAN Notices, pp. 116–127. San Francisco, California, USA (1992).
37. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for java using annotated constraints. In: Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01), volume 36(11), of *ACM SIGPLAN*, pp. 43–55. Tampa, Florida, USA (2001).
38. Ruf, E.: Effective synchronization removal for java. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), vol. 35(5) of *SIGPLAN Notices*, pp. 208–218. Vancouver, British Columbia, Canada (2000).
39. Ruggieri, C., Murtagh, T.P.: Lifetime analysis of dynamically allocated objects. In: 15th ACM Symposium on Principles of Programming Languages (POPL'88), pp. 285–293. San Diego, California, USA (1988).
40. Salcianu, A.: Pointer analysis and its application to java programs. PhD thesis, MIT (2001).
41. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01), volume 36(7) of *SIGPLAN Notices*, pp. 12–23, Snowbird, Utah, USA (2001)
42. Scozzari, F.: Logical optimality of groundness analysis. Theoret. Compu. Sci., **277**(1–2), 149–184 (2002).
43. Søndergaard, H.: An application of abstract interpretation of logic programs: occur check reduction. In: Robinet, B., Wilhelm, R. (eds.), Proc. of the European Symposium on Programming (ESOP), vol. 213 of Lecture Notes in Computer Science, pp. 327–338. Springer, Saarbrücken, Federal Republic of Germany (1986).
44. Spoto, F.: The JULIA generic static analyser. Available at the address `http://www.sci.univr.it/~spoto/julia` (2004).
45. Spoto, F., Jensen, T.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Sys. (TOPLAS), **25**(5), 578–630 (2003).
46. Streckenbach, M., Snelting, G.: Points-to for Java: A General Framework and an Empirical Comparison. Technical report, Universität Passau, Germany (2000).
47. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**, 285–309 (1955).
48. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proc. of OOPSLA'00, vol. 35(10) of SIGPLAN Notices, pp. 281–293. ACM, Minneapolis, Minnesota, USA (2000).
49. Vivien, F., Rinard, M.: Incrementalized pointer and escape analysis. In: Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01), volume 36(5), *SIGPLAN Notices*, pp. 35–46. Snowbird, Utah, USA (2001).
50. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Pugh, W., Chambers, C. (eds.), Proc. of ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04), pp. 131–144. ACM, Washington, DC, USA (2004).
51. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for java programs. In: 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99), volume 34(1), SIGPLAN Notices, pp. 187–206. Denver, Colorado, USA (1999).
52. Winskel, G.: The Formal Semantics of Programming Languages. The MIT Press (1993).