Deriving Escape Analysis by Abstract Interpretation

PATRICIA M. HILL
University of Leeds, United Kingdom
and
FAUSTO SPOTO
Università di Verona, Italy

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 a formally correct escape analyser, able to detect the stack allocatable creation points of Java (bytecode) applications.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*

General Terms: languages, theory

Additional Key Words and Phrases: Semantics, static analysis, abstract interpretation, garbage collection

1. INTRODUCTION

Escape analysis identifies, at compile-time, some run-time data structures which do not *escape* from a given context. It has been studied for functional [Park and Goldberg 1992; Deutsch 1997; Blanchet 1998] as well as for object-oriented languages [Ruggieri and Murtagh 1988; Agrawal 1999; Bogda and Hölzle 1999; Whaley and Rinard 1999; Gay and Steensgaard 2000; Ruf 2000; Streckenbach and Snelting 2000; Rountev et al. 2001; Salcianu and Rinard 2001; Vivien and Rinard 2001; Blanchet 2003; Choi et al. 2003; Whaley and Lam 2004]. 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 at run-time *w.r.t.*

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

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 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 Figure 1. This program defines geometric figures `square` and `circle` that can be rotated. The method `scan` of the class `scan` is used to call a method `rotate` on each figure of a sequence `n` of figures passed as a parameter. This method also creates two new figures and prepends these to the given sequence `n`. 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. Instead, 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 `circle`s rather than `square`s at π_2, and if we assumed that the `scan` method is passed a list of `circle`s 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 [Ruf 2000; Salcianu and Rinard 2001; Whaley and Lam 2004] 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

The first contribution of this paper is a clean construction of an escape analysis through abstract interpretation [Cousot and Cousot 1977; 1992], 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*. Hence, they contain no spurious element. Moreover, it states that the abstract operations make the best possible use of the abstract information expressed by the abstract domains. The second contribution is the definition of *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. These garbage collectors both improve the

```
class figure :
field next : figure
method def() : void is empty
method rot(a : angle) : void is empty
method draw() : void is empty

class square extends figure :
fields side, x, y : int
field rotation : angle
method def() : void is
  this.side := 1;
  this.x, this.y := 0;
  this.rotation := new angle;   {π1}
  this.rotation.degree = 0;
method rot(a : angle) : void is
  this.rotation := a;
method draw() : void is
  % something using this.rotation here...

class circle extends figure :
fields radius, x, y : int
method def() : void is
  this.radius := 1;
  this.x, this.y := 0;          {w0}
method draw() : void is
  % put something here...
```

```
class angle :
field degree : int
method acute() : int is
  out := this.degree < 90;

class scan :
method scan(n : figure) : void is
  f : 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)
    rotate(f);        {w3}
    f := f.next
method rotate(f : figure) : void is
  a : angle;
  a := new angle;     {π4}
  f.rot(a);
  a.degree := 0;
  while (a.degree < 360)
    a.degree := a.degree + 1;
    f.draw();
```

Fig. 1. The running example.

precision and efficiency of the analyses and provide an elegant proof of 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.

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).

From a technical point of view, we achieve our goal through the definition of a basic abstract domain $\mathcal{E}$ which we then formally refine to the abstract domain $\mathcal{ER}$ that we have actually implemented.

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 its actual parameters and any exceptions thrown by m. 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 Figure 1. Assume that nil 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* $E = \{\mathtt{n}\}$*. 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* $\varnothing$*.*

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* (Figure 7). 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 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.

Example 2. *Consider again the method* `scan` *in Figure 1. We propagate the set* $\{\overline{\pi}\}$ *through* `scan`*'s statements by following the translation of high-level statements into the* bytecodes *in Figure 7. These bytecodes are described in Section 3 (a more*

{only `this` of type `scan` is in scope here}
{
  f : figure;

  f := new square; $\{\pi_s\}$
  $\{p_1\}$
  f.def();   {this creates an object at π_1 (Figure 1)}
  $\{p_2\}$
}
$\{p_3\}$

Fig. 2. A program fragment where the set of variables in scope grows and shrinks.

detailed description of part of this execution is presented in Example 34). We get

method scan(n : figure) : void *is*
$\{\overline{\pi}\}$
  f : figure;
$\{\overline{\pi}\}$
  f := new square; $\{\pi_2\}$
$\{\overline{\pi}, \pi_2\}$
  f.next := f;
$\{\overline{\pi}, \pi_2\}$
  f.def();
$\{\overline{\pi}, \pi_1, \pi_2\}$
  rotate(f);
$\{\overline{\pi}, \pi_1, \pi_2, \pi_4\}$
  f := new circle; $\{\pi_3\}$
$\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$
  f.def();
$\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$
  f.next = n;
$\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$
  while(f)
$\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$
    rotate(f);
$\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$
    f := f.next
$\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$.

The restriction of $\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$ *to* $E = \{\texttt{n}\}$ *is* $\{\pi_1, \pi_2, \pi_3, \pi_4\}$ *(objects of class* `scan` *created at* $\overline{\pi}$ *are not compatible with the type of* `n`*), which is a conservative but 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 at the end of each method, $\mathcal{E}$ has very poor computational properties. In terms of abstract interpretation, it is too imprecise and just as for as the basic domain $\mathcal{G}$ for *groundness analysis* of logic programs [Jones and Søndergaard 1987], it needs refining [Armstrong et al. 1998].

Nevertheless, is must be observed that the abstract domain $\mathcal{E}$ already contains some non-trivial computational property. 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 interacts with the late-binding mechanism.* This is what actually happens in Example 2. As another example, note 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 Figure 2, 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 interacts with the static type information.* Namely, the static type information constrains the possible approximations from $\mathcal{E}$ in a given program point, before the actual analysis takes place. Note that, by *static type information* at a given program point, we mean the *type environment i.e.,* the (finite) set of variables that are in scope together with their associated types (Definition 4).

We formalise the fact that the approximation in $\mathcal{E}$ can shrink, by means of an *abstract garbage collector* (Definition 26) *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.* Just as the *concrete* garbage collector claims back the objects which can no more be reached when a variable goes out of scope.

The abstract garbage collector is of no use in Example 2 since, for instance, after the creation point π_3, it would be wrong 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 the abstract domain $\mathcal{E}$ does not distinguish between different variables, so that we do not know whether π_2 was bound to `f`, and hence it must be abstractly garbage collected, or rather to `n`, and hence it must be kept in the approximation. 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 our escape analysis, but it does not include any other related information that may improve the precision of the algorithmic compositional definition of the analysis, such as explicit information about the creation points of the objects bound to *a given* variable or field. The ability to reason on a per variable basis is, in fact, 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 $\mathcal{E}$.

Example 3. *Consider again the method* `scan` *in Figure 1. We start the analysis* *from the element* $[\mathtt{this} \mapsto \{\overline{\pi}\}] \star []$ *of* $\mathcal{ER}$ *which expresses the fact that the variable* `this` *is initially bound to an object created at the external creation point* $\overline{\pi}$ *and all other variables and fields are initially bound to nil (if they have class type) or to an integer (otherwise). The operator* $\star$ *is a pair-separator; its component* $[\mathtt{this} \mapsto \{\overline{\pi}\}]$ *is the approximation for the variables in scope and its component* $[]$ *is the approximation for the fields. The information is then propagated by using the abstract operations in Figure 8 (a more detailed description of part of this execution is presented in Example 52). Then, just as for the domain* $\mathcal{E}$*, at the end of the method the result is restricted to the set* E*:*

method $\mathtt{scan}(\mathtt{n} : \mathtt{figure}) : \mathtt{void}$ *is*
$[\mathtt{this} \mapsto \{\overline{\pi}\}] \star []$
  $\mathtt{f} : \mathtt{figure};$
$[\mathtt{this} \mapsto \{\overline{\pi}\}] \star []$
  $\mathtt{f} := \mathtt{new\ square};\quad \{\pi_2\}$
$[\mathtt{f} \mapsto \{\pi_2\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star []$
  $\mathtt{f.next} := \mathtt{f};$
$[\mathtt{f} \mapsto \{\pi_2\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}]$
  $\mathtt{f.def}();$
$[\mathtt{f} \mapsto \{\pi_2\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1\}]$
  $\mathtt{rotate}(\mathtt{f});$
$[\mathtt{f} \mapsto \{\pi_2\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$
  $\mathtt{f} := \mathtt{new\ circle};\quad \{\pi_3\}$
$[\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$
  $\mathtt{f.def}();$
$[\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$
  $\mathtt{f.next} = \mathtt{n};$
$[\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$
  $\mathtt{while}(\mathtt{f})$
$[\mathtt{f} \mapsto \{\pi_2, \pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$
    $\mathtt{rotate}(\mathtt{f});$
$[\mathtt{f} \mapsto \{\pi_2, \pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$
    $\mathtt{f} := \mathtt{f.next}$
$[\mathtt{this} \mapsto \{\overline{\pi}\}] \star [].$

The restriction of $[\mathtt{this} \mapsto \{\overline{\pi}\}] \star []$ *to* $E = \{\mathtt{n}\}$ *is* $[] \star []$*, which leads to a much more precise approximation* $\varnothing$ *than the set* $\{\pi_1, \pi_2, \pi_3, \pi_4\}$ *of Example 2.*

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 45).

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 $\mathcal{ER}$ can actually be used for a real escape analysis of Java bytecode.

1.4 Structure of the Paper

After a brief summary of our notation and terminology, in Section 3 we recall the watchpoint framework of [Spoto 2001] 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. Section 9 contains the proofs not inlined in the main part of the paper.

Preliminary and partial versions of this paper appeared in [Hill and Spoto 2002a] and [Hill and Spoto 2002b].

2. PRELIMINARIES

A total (partial) function f is denoted by $\mapsto$ ($\rightarrow$). The *domain* (*codomain*) 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 $dom(f) = \{v_1, \ldots, v_n\}$ 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 $f(x) = x$ then x is a *fixpoint* of f. The set of fixpoints of f is denoted by $\mathsf{fp}(f)$.

The two components of a *pair* are separated by $\star$. A definition of S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$. An element x will often stand for the singleton set $\{x\}$, like l in $\lhd_l$ in the definition of $\mathsf{put\_field}$ (Figure 6).

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 (AI) [Cousot and Cousot 1977; 1992]. 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. An abstract operator $\hat{f} : A^n \rightarrow A$ 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 $\hat{f} = \alpha f \gamma$. The *semantics* of a program is the fixpoint of a map $f : C \mapsto C$, where C is the *computational domain.* Its *collecting version* [Cousot and Cousot 1977; 1992] 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.

$$\mathcal{K} = \left\{ \begin{array}{l} \texttt{angle}, \\ \texttt{figure}, \\ \texttt{square}, \\ \texttt{circle}, \\ \texttt{scan} \end{array} \right\} \quad \mathcal{M} = \left\{ \begin{array}{l} \texttt{angle\_acute}, \\ \texttt{figure\_def}, \texttt{figure\_rot}, \texttt{figure\_draw}, \\ \texttt{square\_def}, \texttt{square\_rot}, \texttt{square\_draw}, \\ \texttt{circle\_def}, \texttt{circle\_draw}, \\ \texttt{scan\_scan}, \texttt{scan\_rotate} \end{array} \right\}$$

$$\texttt{square} \le \texttt{figure}, \quad \texttt{circle} \le \texttt{figure} \quad \text{and reflexive cases}$$

$$F(\texttt{angle}) = [\texttt{degree} \mapsto \mathit{int}] \qquad F(\texttt{scan}) = [] \qquad F(\texttt{figure}) = [\texttt{next} \mapsto \texttt{figure}]$$

$$F(\texttt{square}) = \left[\begin{array}{l} \texttt{side} \mapsto \mathit{int}, \texttt{square\_x} \mapsto \mathit{int}, \texttt{square\_y} \mapsto \mathit{int}, \\ \texttt{rotation} \mapsto \texttt{angle}, \texttt{next} \mapsto \texttt{figure} \end{array} \right]$$

$$F(\texttt{circle}) = [\texttt{radius} \mapsto \mathit{int}, \texttt{circle\_x} \mapsto \mathit{int}, \texttt{circle\_y} \mapsto \mathit{int}, \texttt{next} \mapsto \texttt{figure}]$$

$$\begin{aligned}
M(\texttt{angle}) &= [\texttt{acute} \mapsto \texttt{angle_acute}] \\
M(\texttt{figure}) &= [\texttt{def} \mapsto \texttt{figure_def}, \texttt{rot} \mapsto \texttt{figure_rot}, \texttt{draw} \mapsto \texttt{figure_draw}] \\
M(\texttt{square}) &= [\texttt{def} \mapsto \texttt{square_def}, \texttt{rot} \mapsto \texttt{square_rot}, \texttt{draw} \mapsto \texttt{square_draw}] \\
M(\texttt{circle}) &= [\texttt{def} \mapsto \texttt{circle_def}, \texttt{rot} \mapsto \texttt{figure_rot}, \texttt{draw} \mapsto \texttt{circle_draw}] \\
M(\texttt{scan}) &= [\texttt{scan} \mapsto \texttt{scan_scan}, \texttt{rotate} \mapsto \texttt{scan_rotate}]
\end{aligned}$$

$$\begin{aligned}
P(\texttt{angle_acute}) &= [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{angle}] \\
P(\texttt{figure_rot}) &= [\texttt{a} \mapsto \texttt{angle}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{figure}] \\
P(\texttt{scan_rotate}) &= [\texttt{f} \mapsto \texttt{figure}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{scan}] \\
P(\texttt{figure_def}) &= [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{figure}] \text{ (the other cases of } P \text{ are like this)}
\end{aligned}$$

Fig. 3. The static information of the program in Figure 1.

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 [Spoto and Jensen 2003]. It allows us to derive a compositional analyser from a specification of a domain of abstract states and operations which work over them. 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 [Cortesi et al. 1998; Cousot and Cousot 1977; 1992].

3.1 Programs and Creation Points

We use here a more concrete notion of objects than [Spoto and Jensen 2003]. This is because, for escape analysis, every object has to be associated with its *creation point*.

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

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

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

The initial value for a variable of a given type is defined as $\Im(int) = 0$, $\Im(\kappa) = nil$ for $\kappa \in \mathcal{K}$. This function is extended to type environments as $\Im(\tau)(v) = \Im(\tau(v))$ for every $v \in \mathsf{dom}(\tau)$.

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. *An example program is given in Figure 1. Here Id includes, in addition to* `out`, `this`*, user-defined identifiers such as* `rotation`, `def`, `x`, `y`, `rot`*. The set of classes $\mathcal{K}$ is* {`angle`, `figure`, `square`, `circle`, `scan`} *where the ordering $\leq$ is defined by the* `extends` *relation so that* `square` $\leq$ `figure` *and* `circle` $\leq$ `figure`*. Variables for this program include* `x`, `y`*,* `f`*,* `n`*,* `out`*,* `radius` *and* `this`*. At points w_0 and w_1, the type environments are*

$$\begin{aligned}\tau_{w_0} &= [\mathtt{out} \mapsto int, \mathtt{this} \mapsto \mathtt{circle}]\\ \tau_{w_1} &= [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{n} \mapsto \mathtt{figure}, \mathtt{out} \mapsto int, \mathtt{this} \mapsto \mathtt{scan}].\end{aligned}$$

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. *Let $\mathcal{M}$ be a finite set of* methods*. We define*

$$\begin{aligned}Fields &= \{F : \mathcal{K} \mapsto TypEnv \mid \mathtt{this} \notin \mathsf{dom}(F(\kappa)) \text{ for every } \kappa \in \mathcal{K}\}\\ Methods &= \mathcal{K} \mapsto (Id \to \mathcal{M})\\ Pars &= \{P : \mathcal{M} \mapsto TypEnv \mid \{\mathtt{out}, \mathtt{this}\} \subseteq \mathsf{dom}(P(\nu)) \text{ for every } \nu \in \mathcal{M}\}.\end{aligned}$$

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

Definition 7. *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 [Lindholm and Yellin 1999]. For instance, we write `circle_x` for the field `x` of the class `circle`.

Example 8. *The static information of the program in Figure 1 is shown in Figure 3. Note that the result for the method in Figure 1 becomes the type of* `out` *for the corresponding method in Figure 3.*

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. *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. Let* $\pi \in \Pi$, $F \in Fields$ *and* $M \in Methods$. *We define* $F(\pi) = F(k(\pi))$ *and* $M(\pi) = M(k(\pi))$.

Example 10. *Consider again the program in Figure 1. Then* $\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$ *is the set of creation points, where we assume that* $\overline{\pi}$ *decorates a creation point for* `scan` *(*$\overline{\pi}$ *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 *nil*, 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. A memory can be *updated* by assigning new (type consistent) values to the variables in its frames.

Definition 11. *Let Loc be an infinite set of* locations *and* $Value = \mathbb{Z} + Loc + \{nil\}$. *We define* frames, objects *and* memories *as*

$$\begin{aligned}
Frame_\tau &= \left\{ \phi \in \mathsf{dom}(\tau) \mapsto Value \;\middle|\; \begin{array}{l} \textit{for every } v \in \mathsf{dom}(\tau) \\ \quad \textit{if } \tau(v) = int \textit{ then } \phi(v) \in \mathbb{Z} \\ \quad \textit{if } \tau(v) \in \mathcal{K} \textit{ then } \phi(v) \in \{nil\} \cup Loc \end{array} \right\} \\
Obj &= \{\pi \star \phi \mid \pi \in \Pi,\ \phi \in Frame_{F(\pi)}\} \\
Memory &= \{\mu \in Loc \to Obj \mid \mathsf{dom}(\mu) \textit{ is finite}\}.
\end{aligned}$$

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* $\mu_1 \lhd_L \mu_2$, *if* $L \subseteq \mathsf{dom}(\mu_2)$ *and for every* $l \in L$ *we have* $\mu_1(l).\pi = \mu_2(l).\pi$.

Example 12. *Consider again the program in Figure 1 and its static information in Figure 3. A frame, a memory and some objects for this program at program point* w_1 *with locations* $l, l', l'' \in Loc$ *are illustrated in Figure 4.*

Let $\tau_{w_1} = [\texttt{f} \mapsto \texttt{figure}, \texttt{n} \mapsto \texttt{figure}, \texttt{out} \mapsto int, \texttt{this} \mapsto \texttt{scan}]$ *be, as given in Example 5, the type environment at program point* w_1. *Let*

$$\begin{aligned}
\phi_1 &= [\texttt{f} \mapsto l', \texttt{n} \mapsto nil, \texttt{out} \mapsto 2, \texttt{this} \mapsto l] \\
\phi_2 &= [\texttt{f} \mapsto 2, \texttt{n} \mapsto l', \texttt{out} \mapsto -2, \texttt{this} \mapsto l].
\end{aligned}$$

Then $Frame_{\tau_{w_1}}$ *contains* ϕ_1 *(illustrated in Figure 4) but not* ϕ_2 *because* `f` *is bound to* 2 *in* ϕ_2 *(while it has class* `figure` *in* τ_{w_1}*).*

An object o_1 *created at* $\overline{\pi}$ *has class* $k(\overline{\pi}) = \texttt{scan}$ *so that, as* $F(\texttt{scan}) = []$,

$o_1 = \overline{\pi} \star []$.

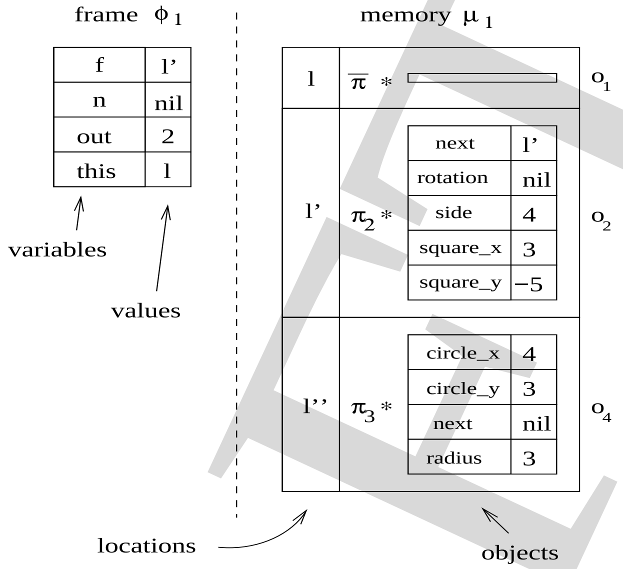

Fig. 4. State $\sigma_1 = \phi_1 \star \mu_1$ for type environment τ_{w_1} (see Example 5) at watchpoint w_1.

Objects created at π_2 *have class* `square` *so that these could be*

$$o_2 = \pi_2 \star [\mathtt{next} \mapsto l', \mathtt{rotation} \mapsto nil, \mathtt{side} \mapsto 4, \mathtt{square\_x} \mapsto 3, \mathtt{square\_y} \mapsto -5]$$
$$o_3 = \pi_2 \star [\mathtt{next} \mapsto nil, \mathtt{rotation} \mapsto l, \mathtt{side} \mapsto 4, \mathtt{square\_x} \mapsto 3, \mathtt{square\_y} \mapsto -5].$$

Similarly, since $k(\pi_3) = \mathtt{circle}$*, an object created at* π_3 *is*

$$o_4 = \pi_3 \star [\mathtt{circle\_x} \mapsto 4, \mathtt{circle\_y} \mapsto 3, \mathtt{next} \mapsto nil, \mathtt{radius} \mapsto 3].$$

The objects o_1, o_2, o_4 *are illustrated in Figure 4.*

With these objects, Memory contains the maps

$$\mu_1 = [l \mapsto o_1, l' \mapsto o_2, l'' \mapsto o_4], \quad \mu_2 = [l \mapsto o_2, l' \mapsto o_1], \quad \mu_3 = [l \mapsto o_2, l' \mapsto o_3].$$

The memory μ_1 *is illustrated in Figure 4.*

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* $\mu_1 \lhd_{l'} \mu_3$ *and* $\mu_2 \lhd_l \mu_3$*. Also, letting* $\mu_4 = [l \mapsto o_1, l' \mapsto o_3, l'' \mapsto o_4]$*, then we have* $\mu_1 \lhd_{\{l,l',l''\}} \mu_4$*.*

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 the type environment. This is a sensible constraint for the memory allocated by strongly-typed languages such as Java [Arnold et al. 2000].

Definition 13. *Let* $\phi \in Frame_\tau$ *and* $\mu \in Memory$*. We say that* ϕ *is* weakly τ-correct *w.r.t.* μ *if for every* $v \in \mathsf{dom}(\phi)$ *such that* $\phi(v) \in Loc$ *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. *Let $\phi \in Frame_\tau$ and $\mu \in Memory$. 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 $F(o.\kappa)$-correct w.r.t. μ.

Example 15. *Let τ_{w_1}, ϕ_1, μ_1, μ_2 and μ_3 be as in Example 12.*

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

$$\begin{gathered}\{v \in \mathsf{dom}(\phi_1) \mid \phi_1(v) \in Loc\} = \{\mathtt{this}, \mathtt{f}\},\\ \{\phi_1(\mathtt{this}), \phi_1(\mathtt{f})\} = \{l, l'\} \subseteq \mathsf{dom}(\mu_1),\\ k(\mu_1(l).\pi) = k(o_1.\pi) = k(\overline{\pi}) = \mathtt{scan} = \tau_{w_1}(\mathtt{this}),\\ k(\mu_1(l').\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{square} \le \mathtt{figure} = \tau_{w_1}(\mathtt{f}).\end{gathered}$$

Condition 2 of Definition 14 holds because

$$\begin{gathered}\mathsf{rng}(\mu_1) = \{o_1, o_2, o_4\},\\ \mathsf{rng}(o_1.\phi) = \mathsf{rng}(o_4.\phi) \cap Loc = \varnothing,\\ \mathsf{rng}(o_2.\phi) \cap Loc = \{l'\} \subseteq \mathsf{dom}(\mu_1),\\ k(\mu_1(l').\pi) = k(o_2.\pi) = \mathtt{square} \le \mathtt{figure} {=} F(o_2.\pi)(\mathtt{next}).\end{gathered}$$

—$\phi_1 \star \mu_2 : \tau_{w_1}$ *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 $\mathtt{square} \not\le \mathtt{scan}$.*

—$\phi_1 \star \mu_3 : \tau_{w_1}$ *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 $F(o_3.\pi)$-correct w.r.t. μ_3, since we have that $o_3.\phi(\mathtt{rotation}) = l$, $\mathtt{square} \not\le \mathtt{angle}$ but $k(\mu_3(l).\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{square}$ and $F(o_3.\pi)(\mathtt{rotation}) = F(\mathtt{square})(\mathtt{rotation}) = \mathtt{angle}$.*

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 *nil*. This condition could be relaxed in Definition 16. This would lead to simplifications in the following sections (like in Definition 26). However, our condition is consistent with the specification of the Java programming language [Arnold et al. 2000]. Note, however, that there is no such hypothesis about the local variable number 0 of the Java Virtual Machine, which stores the `this` object [Lindholm and Yellin 1999].

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

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

Example 17. *With the hypotheses of Example 15, at program point w_1 we have $\phi_1 \star \mu_1 \in \Sigma_{\tau_{w_1}}$, but $\phi_1 \star \mu_2 \notin \Sigma_{\tau_{w_1}}$ and $\phi_1 \star \mu_3 \notin \Sigma_{\tau_{w_1}}$ (since $\phi_1 \star \mu_2 : \tau_{w_1}$ and $\phi_1 \star \mu_3 : \tau_{w_1}$ do not hold, Example 15).*

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

Fig. 5. The signature of the operations over the states.

3.3 The Operations over the Concrete States

Figures 5 and 6 show the signatures and the definitions, respectively, of a set of operations over the concrete states for a type environment τ. These operations can be seen as a simplified set of bytecodes, similar to those of the Java Virtual Machine [Lindholm and Yellin 1999]. The variable *res* holds intermediate results such as the top element of the operand stack of the Java Virtual Machine. We briefly introduce these operations.

—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.

—The put operations store in v the value of *res* or of a field of an object pointed to by *res*. In this second case (put_field) the object whose field is modified must exist in the memory of the state holding the new value of the field. This is expressed by the update relation (Definition 11). As there is no result, *res* is removed.

—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).

—The operation is_nil checks that *res* points to *nil*.

$$
\begin{array}{rlrl}
\mathsf{nop}_\tau(\phi\star\mu) &= \phi\star\mu & \mathsf{get\_int}^i_\tau(\phi\star\mu) &= \phi[res\mapsto i]\star\mu\\
\mathsf{get\_nil}^\kappa_\tau(\phi\star\mu) &= \phi[res\mapsto nil]\star\mu & \mathsf{get\_var}^v_\tau(\phi\star\mu) &= \phi[res\mapsto\phi(v)]\star\mu\\
\mathsf{restrict}^{vs}_\tau(\phi\star\mu) &= \phi|_{-vs}\star\mu,\ \ vs\subseteq\mathsf{dom}(\tau) & \mathsf{expand}^{v:t}_\tau(\phi\star\mu) &= \phi[v\mapsto\Im(t)]\star\mu
\end{array}
$$

$$
\mathsf{put\_var}^v_\tau(\phi\star\mu) = \phi[v\mapsto\phi(res)]|_{-res}\star\mu
$$

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

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

$$
=_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = \begin{cases}\phi_2[res\mapsto 1]\star\mu_2 & \text{if } \phi_1(res)=\phi_2(res)\\ \phi_2[res\mapsto -1]\star\mu_2 & \text{if } \phi_1(res)\neq\phi_2(res)\end{cases}
$$

$$
+_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = \phi_2[res\mapsto\phi_1(res)+\phi_2(res)]\star\mu_2
$$

$$
\mathsf{is\_nil}_\tau(\phi\star\mu) = \begin{cases}\phi[res\mapsto 1]\star\mu & \text{if } \phi(res)=nil\\ \phi[res\mapsto -1]\star\mu & \text{otherwise}\end{cases}
$$

$$
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(\phi\star\mu) = [\iota_1\mapsto\phi(v_1),\ldots,\iota_n\mapsto\phi(v_n),\mathtt{this}\mapsto\phi(res)]\star\mu
$$

$$
\text{where } \{\iota_1,\ldots,\iota_n\} = P(\nu)\setminus\{\mathtt{out},\mathtt{this}\} \text{ (alphabetically ordered)}
$$

$$
\mathsf{return}^\nu_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = \begin{cases}\phi_1[res\mapsto\phi_2(\mathtt{out})]\star\mu_2 & \text{if } L=\mathsf{rng}(\phi_1)|_{-res}\cap Loc \text{ and } \mu_1\lhd_L\mu_2\\ \text{undefined} & \text{otherwise}\end{cases}
$$

$$
\mathsf{new}^\pi_\tau(\phi\star\mu) = \phi[res\mapsto l]\star\mu[l\mapsto\pi\star\Im(F(\pi))],\ \ l\in Loc\setminus\mathsf{dom}(\mu)
$$

$$
\mathsf{lookup}^{m,\nu}_\tau(\phi\star\mu) = \begin{cases}\phi\star\mu & \text{if } \phi(res)\neq nil \text{ and } M((\mu\phi(res)).\pi)(m)=\nu\\ \text{undefined} & \text{otherwise}\end{cases}
$$

$$
\mathsf{is\_true}_\tau(\phi\star\mu) = \begin{cases}\phi|_{-res}\star\mu & \text{if } \phi(res)\geq 0\\ \text{undefined} & \text{otherwise}\end{cases}
$$

$$
\mathsf{is\_false}_\tau(\phi\star\mu) = \begin{cases}\phi|_{-res}\star\mu & \text{if } \phi(res)<0\\ \text{undefined} & \text{otherwise.}\end{cases}
$$

Fig. 6. The operations over concrete states.

—The operation call (return) is used before (after) a call to a method ν. While call^ν creates a new state in which ν can execute, the operation return^ν restores the state σ which was current before the call to ν, and stores in res the result of the call. The update relation (Definition 11) requires that the variables of σ have not been changed during the execution of the method (although their fields 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 $\mathsf{lookup}^{m,\nu}$ checks if, by calling the method identified by m of the object o pointed to by res, the method ν is run. This depends on the class $k((\mu\phi(res)).\pi)$ of o.

—The operation is_true (is_false) checks if res contains true (false).

Example 18. *Consider again the example in Figure 1. Let $\tau = \tau_{w_1}$, ϕ_1, μ_1 and $\sigma_1 = \phi_1 \star \mu_1$ be as in Figure 4 so that, as indicated in Example 17, the σ_1 could be the current state at program point w_1. The computation continues as follows [Spoto and Jensen 2003].*

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

Let o_1, o_2, o_3, o_4 be as in Example 12. Then

$$\begin{aligned}\sigma_2 &= \phi_1\left[res \mapsto \phi_1(f)\right] \star \mu_1\\ &= \phi_1\left[res \mapsto l'\right] \star \mu_1\\ &= \left[\ \mathtt{f} \mapsto l', \mathtt{n} \mapsto nil, \mathtt{out} \mapsto 2, res \mapsto l', \mathtt{this} \mapsto l\ \right] \star \mu_1.\end{aligned}$$

The lookup *operations determine which is the target of the first virtual call* f.def() *in Figure 1. As a result only one of the following blocks of code is run depending on which* lookup *check is defined.*

$$\begin{array}{rll}
\sigma'_3 = & \mathsf{lookup}^{\mathtt{def},\mathtt{figure_def}}_{\tau[res \mapsto \mathtt{figure}]}(\sigma_2) & \textit{select } \mathtt{figure_def}\\
\sigma'_4 = & \mathsf{call}^{\mathtt{figure_def}}_{\tau[res \mapsto P(\mathtt{figure_def})(\mathtt{this})]}(\sigma'_3) & \textit{initialise } \mathtt{figure_def}\\
\sigma'_5 = & \textit{the final state of } \mathtt{figure_def} \textit{ from } \sigma'_4 & \\
\sigma'_6 = & \mathsf{return}^{\mathtt{figure_def}}_{\tau[res \mapsto P(\mathtt{figure_def})(\mathtt{this})]}(\sigma'_3)(\sigma'_5) & \textit{return to the caller}\\
\\
\sigma''_3 = & \mathsf{lookup}^{\mathtt{def},\mathtt{square_def}}_{\tau[res \mapsto \mathtt{figure}]}(\sigma_2) & \textit{select } \mathtt{square_def}\\
\sigma''_4 = & \mathsf{call}^{\mathtt{square_def}}_{\tau[res \mapsto P(\mathtt{square_def})(\mathtt{this})]}(\sigma''_3) & \textit{initialise } \mathtt{figure_def}\\
\sigma''_5 = & \textit{the final state of } \mathtt{square_def} \textit{ from } \sigma''_4 & \\
\sigma''_6 = & \mathsf{return}^{\mathtt{square_def}}_{\tau[res \mapsto P(\mathtt{square_def})(\mathtt{this})]}(\sigma''_3)(\sigma''_5) & \textit{return to the caller}\\
\\
\sigma'''_3 = & \mathsf{lookup}^{\mathtt{def},\mathtt{circle_def}}_{\tau[res \mapsto \mathtt{figure}]}(\sigma_2) & \textit{select } \mathtt{circle_def}\\
\sigma'''_4 = & \mathsf{call}^{\mathtt{circle_def}}_{\tau[res \mapsto P(\mathtt{circle_def})(\mathtt{this})]}(\sigma'''_3) & \textit{initialise } \mathtt{circle_def}\\
\sigma'''_5 = & \textit{the final state of } \mathtt{circle_def} \textit{ from } \sigma'''_4 & \\
\sigma'''_6 = & \mathsf{return}^{\mathtt{circle_def}}_{\tau[res \mapsto P(\mathtt{circle_def})(\mathtt{this})]}(\sigma'''_3)(\sigma'''_5) & \textit{return to the caller.}
\end{array}$$

For each lookup *operation, we have $(\sigma_2.\phi)(res) = l' \neq nil$ 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 $k(\pi_2) =$* square. *Moreover M(square)(def) = square_def (Figure 3). So, the only* lookup *operation which is defined 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

$$\sigma''_4 = [\mathtt{this} \mapsto l'] \star \mu_1.$$

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 the final state of this execution which is*

$$\sigma''_5 = [\mathtt{out} \mapsto 0] \star \underbrace{[l \mapsto o_1, l' \mapsto o_5, l'' \mapsto o_4]}_{\mu_5},$$

where

$$o_5 = \pi_2 \star [\texttt{next} \mapsto l', \texttt{side} \mapsto 1, \texttt{square\_x} \mapsto 0, \texttt{square\_y} \mapsto 0, \texttt{rotation} \mapsto o_6],$$
$$o_6 = \pi_1 \star [\texttt{degree} \mapsto 0].$$

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 in the res variable of the caller. Namely,*

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

The next section, which is concerned with an approximation of the concrete semantics, will use the powerset $\wp(\Sigma_\tau)$ to denote a *set of* possible concrete states. Note that dealing with powersets means that the semantics becomes non-deterministic. For instance, in Example 18 more than one target of the $\texttt{f.def}()$ 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). We will assume the result, proved in [Spoto and Jensen 2003], that every abstraction of $\wp(\Sigma_\tau)$, $\cup$, and of the powerset extension of the operations in Figure 6 induces an abstraction of the concrete semantics.

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. Hence it can be defined without any consideration about name clashes or multiple instances of the same variable during recursion. 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.

EXAMPLE 19. *Consider the program in Figure 1. Are there any objects created at π_4 and reachable at program point w_1? The type environment at w_1, which is $\tau_{w_1} = [\texttt{f} \mapsto \texttt{figure}, \texttt{n} \mapsto \texttt{figure}, \texttt{out} \mapsto int, \texttt{this} \mapsto \texttt{scan}]$, 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* $\texttt{square} \leq \texttt{figure}$*, 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* $\texttt{f.draw}()$.

The reasoning in Example 19 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. Before giving this definition we need a guarantee that Definition 21 is well-defined. This result states that the frame of an object o in memory is itself a state for the instance variables of o.

LEMMA 20. *Let $\phi \star \mu \in \Sigma_\tau$ and $o \in \mathsf{rng}(\mu)$. Then $(o.\phi) \star \mu \in \Sigma_{F(o.\pi)}$.*

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 $F(o.\pi)$-correct *w.r.t.* μ so that $(o.\phi) \star \mu : F(o.\pi)$. Since `this` $\notin \mathsf{dom}(F(o.\pi))$ (Definition 6) we conclude that $(o.\phi) \star \mu \in \Sigma_{F(o.\pi)}$. $\square$

DEFINITION 21. *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*

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

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

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

Observe that variables and fields of type *int* do not contribute to O_τ.

EXAMPLE 22. *Let ϕ_1, μ_1, $\sigma_1 = \phi_1 \star \mu_1$, o_1 and o_2 be as defined in Figure 4. 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, using the static information in Figure 3, $F(o_1.\pi) = \varnothing$ and $\mathsf{dom}(F(o_2.\pi)) = \{\mathtt{next}, \mathtt{rotation}, \mathtt{side}, \mathtt{square\_x}, \mathtt{square\_y}\}$. From Figure 4 we have $\{\mu_1(o_2.\phi(f)) \mid f \in \mathsf{dom}(F(o_2.\pi)),\ o_2.\phi(f) \in Loc\} = \{o_2\}$ and therefore*

$$O^1_{F(o_2.\pi)}(o_2.\phi \star \mu_1) = \{\mu(o_2.\phi(f)) \mid f \in \mathsf{dom}(F(o_2.\pi)),\ o_2.\phi(f) \in Loc\} = \{o_2\}$$

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.

We can now define the abstraction map for $\mathcal{E}$. It selects the creation points of the reachable objects.

DEFINITION 23. *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 24. *Consider again Example 22. Then, 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 23 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 25. *Consider the program point w_0 in Figure 1 and its type environment $\tau_{w_0} = [\mathtt{out} \mapsto int, \mathtt{this} \mapsto \mathtt{circle}]$. Then $\alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma) \neq \{\pi_4\}$ for every*

$\sigma = \phi \star \mu \in \Sigma_\tau$. *This is because*

$$\alpha^{\mathcal{E}}_{\tau_{w_0}}(\sigma) = \bigcup \left\{ \{o.\pi\} \cup \alpha^{\mathcal{E}}_{F(o.\pi)}((o.\phi) \star \mu) \,\middle|\, v \in \{\texttt{this}\},\ \phi(v) \in Loc,\ o = \mu\phi(v) \right\}.$$

By Definition 13 we know that if $\phi(v) \in Loc$ *then* $k(o.\pi) = \texttt{circle}$. *Hence* $o.\pi = \pi_3$. *We conclude that whether* $\phi(v) = nil$, *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 25 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 26).

Let $e \in \wp(\Pi)$. Then $\delta_\tau(S)$ 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 26. *Let* $S \subseteq \Pi$. *We define* $\delta_\tau(e) = \cup\{\delta^i_\tau(e) \mid i \geq 0\}$ *with*

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

$$\delta^{i+1}_\tau(e) = \begin{cases} \varnothing & \textit{if } \texttt{this} \in \mathsf{dom}(\tau) \textit{ and there is no } \pi \in S \textit{ s.t. } k(\pi) \leq \tau(\texttt{this}) \\ \cup\left\{ \{\pi\} \cup \delta^i_{F(\pi)}(e) \mid \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in e,\ k(\pi) \leq \kappa \right\} & \textit{otherwise.} \end{cases}$$

It follows from Definition 26 that $\delta^i_\tau \subseteq \delta^{i+1}_\tau$ and hence $\delta_\tau = \delta^{\#\Pi}_\tau$. Note that in Definition 26 we consider all subclasses of κ (Example 19).

Example 27. *Consider the program point* w_0 *in Figure 1 and its type environment* $\tau_{w_0} = [\texttt{out} \mapsto int, \texttt{this} \mapsto \texttt{circle}]$. *Let* $e = \{\overline{\pi}, \pi_1, \pi_3, \pi_4\}$. *Then, it can be seen in Figure 3 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*

$$\begin{aligned} \delta^1_{F(\texttt{circle})}(e) &= \cup\left\{ \{\pi\} \cup \delta^0_{F(\pi)}(e) \mid \pi \in e,\ k(\pi) \leq \texttt{figure} \right\} = \{\pi_3\} \\ \delta^{i+1}_{F(\texttt{circle})}(e) &= \cup\left\{ \{\pi\} \cup \delta^i_{F(\pi)}(e) \mid \pi \in e,\ k(\pi) \leq \texttt{figure} \right\}) \\ &= \{\pi_3\} \cup \delta^i_{F(\texttt{circle})(e)}\ , \end{aligned}$$

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

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

Consider the program point w_1 *in Figure 1 and its type environment* $\tau_{w_1} = [\texttt{f} \mapsto \texttt{figure}, \texttt{n} \mapsto \texttt{figure}, \texttt{out} \mapsto int, \texttt{this} \mapsto \texttt{scan}]$. *Let* $e = \{\overline{\pi}, \pi_1, \pi_3, \pi_4\}$. *Suppose* $i > 0$. *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* $i > 0$*,*

$$\begin{aligned}
\delta_{\tau_{w_1}}(e) &= \delta^{i+1}_{\tau_{w_1}}(e) \\
&= \cup\big\{\{\pi\} \cup \delta^i_{F(\pi)}(e) \;\big|\; \kappa \in \{\texttt{figure}, \texttt{scan}\},\ \pi \in e,\ k(\pi) \le \kappa\big\} \\
&= \{\overline{\pi}, \pi_3\} \cup \delta^{i+1}_{F(\texttt{square})}(e) \cup \delta^i_{F(\texttt{circle})}(e) \cup \delta^i_{F(\texttt{scan})}(e) \\
&= \{\overline{\pi}, \pi_3\} \cup \{\pi_1, \pi_3, \pi_4\} \cup \{\pi_3\} \cup \varnothing \\
&= \{\overline{\pi}, \pi_1, \pi_3, \pi_4\}.
\end{aligned}$$

Then all the creation points in e *are useful in* w_1 *(compare this with Example 19).*

Proposition 28 proves 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 28. *Let* $i \in \mathbb{N}$*. The maps* δ^i_τ *and* δ_τ *are lco's.*

Proof. Since $\delta_\tau = \delta^{\#\Pi}_\tau$, it is enough to prove the result for δ^i_τ only. By Definition 26, the maps δ^i_τ for $i \in \mathbb{N}$ are reductive and monotonic. We prove idempotency by induction over $i \in \mathbb{N}$. Let $e \subseteq \Pi$. We have $\delta^0_\tau\delta^0_\tau(e) = \delta^0_\tau(\varnothing) = \varnothing = \delta^0_\tau(e)$. Assume that the result holds for a given $i \in \mathbb{N}$. If $\texttt{this} \in \mathsf{dom}(\tau)$ and there is no $\pi \in e$ such that $k(\pi) \le \tau(\texttt{this})$, then $\delta^i_\tau\delta^i_\tau(e) = \delta^i_\tau(\varnothing) = \varnothing = \delta^i_\tau(e)$. Suppose now that, if $\texttt{this} \in \mathsf{dom}(\tau)$, then there exists $\pi \in e$ such that $k(\pi) \le \tau(\texttt{this})$. By reductivity, $\delta^{i+1}_\tau\delta^{i+1}_\tau(e) \subseteq \delta^{i+1}_\tau(e)$. We prove that the converse inclusion holds. We have

$$\delta^{i+1}_\tau\delta^{i+1}_\tau(e) = \cup\left\{\{\pi\} \cup \delta^i_{F(\pi)}\delta^{i+1}_\tau(e) \;\middle|\; \begin{array}{l}\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ \pi \in \delta^{i+1}_\tau(e),\ k(\pi) \le \kappa\end{array}\right\}. \tag{1}$$

Let $\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K}$ and $\pi \in \Pi$ be such that $k(\pi) \le \kappa$. If $\pi \in \delta^{i+1}_\tau(e)$ then, by reductivity, we have $\pi \in e$. Conversely, if $\pi \in e$ then, by Definition 26, $\pi \in \delta^{i+1}_\tau(e)$. We conclude from (1) that

$$\begin{aligned}
\delta^{i+1}_\tau\delta^{i+1}_\tau(e) &= \cup\big\{\,\{\pi\} \cup \delta^i_{F(\pi)}\delta^{i+1}_\tau(e) \;\big|\; \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in e,\ k(\pi) \le \kappa\,\big\} \\
\text{(monotonicity)} &\supseteq \cup\big\{\,\{\pi\} \cup \delta^i_{F(\pi)}\delta^i_{F(\pi)}(e) \;\big|\; \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in e,\ k(\pi) \le \kappa\,\big\} \\
\text{(ind. hypothesis)} &= \cup\big\{\,\{\pi\} \cup \delta^i_{F(\pi)}(e) \;\big|\; \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in e,\ k(\pi) \le \kappa\,\big\} \\
&= \delta^{i+1}_\tau(e).
\end{aligned}$$

□

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 29 (which can be found in Section 9) 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 29. *We have* $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$ *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 29 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 29 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 (Subsection 4.2).

Definition 30. *Our basic domain for escape analysis is* $\mathcal{E}_\tau = \mathsf{fp}(\delta_\tau)$, *ordered by set inclusion.*

Example 31. *Let* $\tau_{w_0} = [\mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{circle}]$ *and* $\tau_{w_1} = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{n} \mapsto \mathtt{figure}, \mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{scan}]$. *Then*

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

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 23, we know that $\alpha_\tau^{\mathcal{E}}$ is strict and additive and, by Proposition 29, onto $\mathcal{E}_\tau$. Thus, by a general result of abstract interpretation (Section 2), we have the following proposition.

Proposition 32. *The map* $\alpha_\tau^{\mathcal{E}}$ *(Definition 23) is the abstraction map of a Galois insertion from* $\wp(\Sigma_\tau)$ *to* $\mathcal{E}_\tau$.

Note that if, in Definition 30, we had defined $\mathcal{E}_\tau$ as $\wp(\Pi)$, the map $\alpha_\tau^{\mathcal{E}}$ would induce just a Galois connection instead of a Galois insertion, as a consequence of Proposition 29.

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 7 defines the abstract counterparts of the concrete operations in Figure 6. Proposition 33 shows that they are correct and optimal, in the sense of abstract interpretation (Section 2). Optimality is proved by explicitly constructing, for every abstract input e, a set of concrete states X such that the abstraction of the application of the concrete operation to X coincides with the abstract operation shown in Figure 7, applied to e (see Section 9).

Proposition 33. *The operations in Figure 7 are the optimal counterparts induced by* $\alpha^{\mathcal{E}}$ *of the operations in Figure 6 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 Figure 7 coincide with the identity map. This is a sign of the algorithmic 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 $\mathsf{put\_var}$ operation removes variable v from the scope (Figure 6), its abstract counterpart in Figure 7 calls the garbage collector. The same happens for $\mathsf{restrict}$ which, however, removes a *set* of variables from scope. There are also some operations ($\mathsf{is\_nil}$, $\mathsf{put\_field}$, lookup) that use *res* as a temporary variable and one operation ($\mathsf{get\_field}$) that changes the type of res. Hence these abstract operations

$$\begin{aligned}
\mathsf{nop}_\tau(e) &= e & \mathsf{get_int}^i_\tau(e) &= e\\
\mathsf{get_nil}^\kappa_\tau(e) &= e & \mathsf{get_var}^v_\tau(e) &= e\\
\mathsf{is_true}_\tau(e) &= e & \mathsf{is_false}_\tau(e) &= e\\
\mathsf{put_var}^v_\tau(e) &= \delta_{\tau|_{-v}}(e) & \mathsf{is_nil}_\tau(e) &= \delta_{\tau|_{-res}}(e)\\
\mathsf{new}^\pi_\tau(e) &= e \cup \{\pi\} & =_\tau(e_1)(e_2) &= +_\tau(e_1)(e_2) = e_2\\
\mathsf{expand}^{v:t}_\tau(e) &= e & \mathsf{restrict}^{vs}_\tau(e) &= \delta_{\tau_{-vs}}(e)\\
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(e) &= \delta_{\tau|_{\{v_1,\ldots v_n,res\}}}(e) & \cup_\tau(e_1)(e_2) &= e_1 \cup e_2
\end{aligned}$$

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

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

$$\mathsf{return}^\nu_\tau(e_1)(e_2) = \cup\{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K},\ \pi \in e_1,\ k(\pi) \le \kappa\} \cup e_2$$

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

Fig. 7. The optimal abstract operations over $\mathcal{E}$.

also need to call the garbage collector. Note that the definitions of 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 nil*. 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 variables in scope before the call might have been modified by the call. This is reflected by the use of $\delta_{F(\pi)}(\Pi)$ in return, which plays the role of a worst-case assumption on the content of the fields. After Example 34 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 $\delta_{F(\pi)}$ 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 Figure 7 by the more imprecise but simpler definitions:

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

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

EXAMPLE 34. *Let us mimic, in $\mathcal{E}$, the concrete computation of Example 18. Let $\tau = \tau_{w_1}$ be as given in Example 5. We start by constructing elements e_1 and e_2 of $\mathcal{E}$ corresponding to the concrete states σ_1 and σ_2 there. The abstract state e_1 is obtained by abstracting σ_1 (see Example 24):*

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

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.*

$$
\begin{aligned}
e'_3 &= \mathsf{lookup}^{\mathtt{def},\mathtt{figure\_def}}_{\tau[res\mapsto\mathtt{figure}]}(e_2) = \varnothing\\
&\quad \textit{since } \{\pi \in e_2 \mid k(\pi) \le \mathtt{figure} \textit{ and } M(\pi)(\mathtt{def}) = \mathtt{figure\_def}\} = \varnothing\\
e''_3 &= \mathsf{lookup}^{\mathtt{def},\mathtt{square\_def}}_{\tau[res\mapsto\mathtt{figure}]}(e_2)\\
&= \delta_\tau(e_2) \cup \left(\bigcup \left\{ \{\pi\} \cup \delta_{F(\pi)}(e_2) \,\middle|\, \pi \in e'' \right\}\right) = \{\overline{\pi}, \pi_2\} \cup \{\pi_2\}\\
&= \{\overline{\pi}, \pi_2\}\ ,\\
&\quad \textit{since } e'' = \{\pi \in e_2 \mid k(\pi) \le \mathtt{figure} \textit{ and } M(\pi)(\mathtt{def}) = \mathtt{square\_def}\} = \{\pi_2\}\\
e'''_3 &= \mathsf{lookup}^{\mathtt{def},\mathtt{circle\_def}}_{\tau[res\mapsto\mathtt{figure}]}(e_2) = \varnothing\\
&\quad \textit{since } e''' = \{\pi \in e_2 \mid k(\pi) \le \mathtt{figure},\ M(\pi)(\mathtt{def}) = \mathtt{circle\_def}\} = \varnothing.
\end{aligned}
$$

The lookup *operations for* `figure` *and* `circle` *return $\varnothing$ so that, as the abstract operations over $\mathcal{E}$ are strict on $\varnothing$ (Proposition 33), $e'_4 = e'_5 = e'_6 = e'''_4 = e'''_5 = e'''_6 = \varnothing$. 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:*

$$e''_4 = \mathsf{call}^{\mathtt{square\_def}}_{\tau[res\mapsto\mathtt{square}]}(e''_3) = \delta_{[res\mapsto\mathtt{square}]}(e''_3) = \{\pi_2\}.$$

Since $P(\mathtt{square\_def})|_{\mathsf{out}} = [\mathtt{out} \mapsto int]$ and $\mathcal{E}_{[\mathtt{out}\mapsto int]} = \{\varnothing\}$, we do not need to execute the method `square_def` *to conclude that*

$$e''_5 = \varnothing.$$

Hence

$$
\begin{aligned}
e''_6 &= \mathsf{return}^{\mathtt{square\_def}}_{\tau[res\mapsto\mathtt{square}]}(e''_3)(e''_5)\\
&= \cup\{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \pi \in \{\overline{\pi}, \pi_2\}\} = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}.
\end{aligned}
$$

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 Figure 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 34, 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 `this` *could* be modified. For instance, an `angle` could be bound to the field `rotation` of `this`. This is what actually happens for π_1 in the method `square_def` (Figure 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 34, we can only have $e = \varnothing$. When we return to the caller, the actual parameters return into scope. The definition of `return` in Figure 6 shows that these parameters are unchanged (because of the condition $\mu_1 \lhd_{\mathsf{rng}(\phi_1)|_{-res} \cap Loc}\, \mu_2$ on the concrete operation). This is reflected by the condition $\pi \in e_1$ in the abstract `return` operation in Figure 7. 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 53). By using this technique, we can actually improve the precision of the computation in Example 34. 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 35. *Let the approximation provided by $\mathcal{E}$ before the* `f := new circle` *statement in Figure 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:*

$$\begin{aligned}\mathsf{new}^{\pi_3}_{\tau}(\{\overline{\pi}, \pi_1, \pi_2, \pi_4\}) &= \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}\\ \mathsf{put\_var}^{\mathtt{f}}_{\tau[res \mapsto \mathtt{circle}]}(\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}) &= \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}.\end{aligned}$$

This time, the imprecision is a consequence of the fact that $\mathtt{n} \in \mathsf{dom}(\tau)$, $\tau(\mathtt{n}) = \mathtt{figure}$ and $k(\pi_2) = \mathtt{square} \le \mathtt{figure}$. 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 59). 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 35). 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 $*$, which approximates the integers, or a set $e \subseteq \Pi$, which approximates *nil* 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 36. *Let* $\mathit{Value}^{\mathcal{ER}} = \{*\} \cup \wp(\Pi)$. *We define*

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

Example 37. *As in Example 5, let* $\tau_{w_1} = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{n} \mapsto \mathtt{figure}, \mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{scan}]$. *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* $\mathtt{scan} \not\le \mathtt{figure}$.

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

Definition 38. *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 39. *Consider the state* σ_1 *in Figure 4. 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* is well-defined.

Definition 40. *We define* $\widetilde{\tau} = \cup\{F(\kappa) \mid \kappa \in \mathcal{K}\}$. *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* $v \in \mathsf{dom}(\widetilde{\tau})$,

$$\widetilde{\phi}(v) = \begin{cases} \phi(v) & \text{if } v \in \mathsf{dom}(\tau) \\ \Im(\widetilde{\tau}(v)) & \text{otherwise (Definition 4).} \end{cases}$$

Example 41. *Consider the map* F *in Figure 3 for the program in Figure 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 $\phi = [\texttt{circle\_x} \mapsto 12, \texttt{circle\_y} \mapsto 5, \texttt{next} \mapsto l, \texttt{radius} \mapsto 5] \in F(\texttt{circle})$, *with* $l \in Loc$. *We have*

$$\widetilde{\phi} = \begin{bmatrix} \texttt{circle\_x} \mapsto 12, \texttt{circle\_y} \mapsto 5, \texttt{degree} \mapsto 0, \texttt{next} \mapsto l, \texttt{radius} \mapsto 5 \\ \texttt{rotation} \mapsto nil, \texttt{side} \mapsto 0, \texttt{square\_x} \mapsto 0, \texttt{square\_y} \mapsto 0 \end{bmatrix}.$$

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 42. *Let* $Memory^{\mathcal{ER}} = Frame^{\mathcal{ER}}_{\widetilde{\tau}}$. *We define the map*

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

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

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

EXAMPLE 43. *Consider the state* σ_1 *in Figure 4. Let* $\tau = \tau_{w_1}$ *be as given in Example 5. In Example 22 we have shown that* $O_\tau(\sigma_1) = \{o_1, o_2\}$ *and in Example 39 we have computed the value of* $\varepsilon_\tau(\sigma_1)$. *We have*

$$\widetilde{o_1.\phi} = \begin{bmatrix} \texttt{circle\_x} \mapsto 0, \texttt{circle\_y} \mapsto 0, \texttt{degree} \mapsto 0 \\ \texttt{next} \mapsto nil, \texttt{radius} \mapsto 0, \texttt{rotation} \mapsto nil \\ \texttt{side} \mapsto 0, \texttt{square\_x} \mapsto 0, \texttt{square\_y} \mapsto 0 \end{bmatrix},$$
$$\widetilde{o_2.\phi} = \begin{bmatrix} \texttt{circle\_x} \mapsto 0, \texttt{circle\_y} \mapsto 0, \texttt{degree} \mapsto 0 \\ \texttt{next} \mapsto l', \texttt{radius} \mapsto 0, \texttt{rotation} \mapsto nil \\ \texttt{side} \mapsto 4, \texttt{square\_x} \mapsto 3, \texttt{square\_y} \mapsto -5 \end{bmatrix}$$

in $Memory^{\mathcal{ER}}$. *Then (the fields not represented are implicitly bound to* $*$*)*

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

Compare Examples 43 and 24. 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 25), the following example shows that the map $\alpha^{\mathcal{ER}}_{\tau}$ is not necessarily onto.

EXAMPLE 44. *Let* $\tau = [\texttt{c} \mapsto \texttt{circle}]$. *A* `circle` *has no field called* `rotation`. *Then there is no* $\sigma \in \Sigma_\tau$ *such that* $\alpha^{\mathcal{ER}}_{\tau}(\sigma) = [\texttt{c} \mapsto \{\pi_3\}] \star [\texttt{next} \mapsto \varnothing, \texttt{rotation} \mapsto \{\pi_1\}, \ldots]$, *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 approximation of the fields of the objects created at r is not garbage collected by ξ. The approximations of the other fields are garbage collected instead.

Definition 45. *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* $\rho_\tau(s) = \cup\{\rho_\tau^i(s) \mid i \ge 0\}$, *where*

$$\rho_\tau^0(\phi \star \mu) = \varnothing$$
$$\rho_\tau^{i+1}(\phi \star \mu) = \bigcup \left\{ \{\pi\} \cup \rho_{F(\pi)}^i(\phi|_{\mathsf{dom}(F(\pi))} \star \mu) \,\middle|\, v \in \mathsf{dom}(\tau),\ \pi \in \phi(v) \right\}$$

and

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

Example 46. *In Example 44, 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.*

The following property is expected to hold for a garbage collector. Compare Propositions 28 and 47.

Proposition 47. *The map* ξ_τ *is an lco.*

Proof. By Definition 45, the map ξ_τ is reductive and monotonic. For idempotency, we have $\xi_\tau\xi_\tau(\bot) = \bot = \xi_\tau(\bot)$. Let $s \in \mathit{Frame}_\tau^{\mathcal{ER}} \times \mathit{Memory}^{\mathcal{ER}}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and $\phi(\mathtt{this}) = \varnothing$ then $\xi_\tau\xi_\tau(s) = \bot = \xi_\tau(\bot)$. Otherwise, we prove that $\rho_\tau\xi_\tau(s) = \rho_\tau(s)$, which entails the thesis by Definition 45. We have

$$\begin{aligned} \rho_\tau\xi_\tau(\phi \star \mu) &= \rho_\tau(\phi \star \cup\{\mu|_{\mathsf{dom}(F(\pi'))} \mid \pi' \in \rho_\tau(\phi \star \mu)\}) \\ &= \{\pi \in \phi(v) \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K}\} \cup \\ &\qquad \cup \{\pi \in \mu(f) \mid \pi' \in \rho_\tau(\phi \star \mu),\ f \in \mathsf{dom}(F(\pi')),\ F(\pi')(f) \in \mathcal{K}\} \\ &= \rho_\tau(\phi \star \mu). \end{aligned}$$

□

The map 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 29 and 48.

Proposition 48. *We have* $\mathsf{fp}(\xi_\tau) = \mathsf{rng}(\alpha_\tau^{\mathcal{ER}})$.

Proposition 48 (whose proof can be found in Section 9) allows us to assume that $\alpha_\tau^{\mathcal{ER}} : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\xi_\tau)$ and justifies the following definition.

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

$$=_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = +_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = \phi_2\star\mu_2$$

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

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

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

$$\mathsf{return}^\nu_\tau(\phi_1\star\mu_1)(\phi_2\star\mu_2) = \xi_{\tau|_{-res}}(\phi_1|_{-res}\star\mu^\top)\cup([res\mapsto\phi_2(\mathtt{out})]\star\mu_2)$$
$$\text{where } \mu^\top \text{ is the top of } Memory^{\mathcal{ER}}$$

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

Fig. 8. The abstract operations over $\mathcal{ER}$.

Definition 49. *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 38 and 42 we know that the map $\alpha^{\mathcal{ER}}_\tau$ is strict and additive. By Proposition 48 we know that it is onto. Thus we have the following result corresponding to Proposition 32 for the domain $\mathcal{E}$.

Proposition 50. *The map $\alpha^{\mathcal{ER}}_\tau$ 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 Figure 6. Since $\mathcal{ER}$ approximates every variable and field with an abstract value, those abstract operations are similar to those of the domain for *class analysis* in [Palsberg and Schwartzbach 1991] as formulated in [Spoto and Jensen 2003]. However, $\mathcal{ER}$ observes the fields of just the reachable objects (Definition 42), while Palsberg and Schwartzbach's domain observes the fields of all objects in memory.

We define abstract counterparts of the concrete operations in Figure 6 on the abstract domain $\mathcal{ER}$ in the Figure 8. These operations are implicitly strict on $\bot$ except for $\cup$. In this case, we define $\bot \cup (\phi \star \mu) = (\phi \star \mu) \cup \bot = \phi \star \mu$.

PROPOSITION 51. *The operations in Figure 8 are the optimal counterparts induced by $\alpha^{\mathcal{ER}}$ of the operations in Figure 6 and of $\cup$, except for* put_field *which is only proved to be correct.*

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 $\mathcal{ER}$ does not approximate 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 36). The concrete operation get_nil loads *nil* into *res* and hence its abstract counterpart approximates *res* with $\varnothing$. The operation 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_nil operation checks if *res* contains *nil* 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$^\pi$ operation binds *res* to an object created at π. The 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 *nil*. The get_field^f operation returns $\bot$ if it is *always* applied to states where the receiver *res* is *nil*. This is because $\bot$ is the best approximation of the empty set of final states. If, instead, the receiver is not necessarily *nil*, 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 put_field^f operation, we first consider the case when the receiver is always *nil*, in which 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 like `a.g.f` $=$ `m(a)` where the method call `m(a)` resets the field `g` of `a` to *nil*. Since we assume that the left-hand side is evaluated before the right-hand side, the receiver is not necessarily *nil*, but the field updates might not be observable if `a.g.f` is not reachable in any other way but from `a`. In the third and final case for put_field we consider the standard situation when we write into a reachable field of a non-*nil* 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 53). 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 52. *As in Example 34 for $\mathcal{E}$, let us mimic, in $\mathcal{ER}$, the concrete computation of Example 18. We start from the abstraction (Definition 42) of σ_1, given in Example 43. Variables and fields not shown are implicitly bound to $*$ if they have class type and to $*$ if they have type int.*

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

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

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.*

$$\begin{aligned}
s'_3 &= \mathsf{lookup}_{\tau[res\mapsto\mathtt{figure}]}^{\mathtt{def},\mathtt{figure_def}}(s_2) = \bot \\
&\quad \textit{since } e' = \{\pi \in \{\pi_2\} \mid M(\pi)(\mathtt{def}) = \mathtt{figure_def}\} = \varnothing, \\
s''_3 &= \mathsf{lookup}_{\tau[res\mapsto\mathtt{figure}]}^{\mathtt{def},\mathtt{square_def}}(s_2) = \xi_\tau(s_2) = s_2 \\
&\quad \textit{since } e'' = \{\pi \in \{\pi_2\} \mid M(\pi)(\mathtt{def}) = \mathtt{square_def}\} = \{\pi_2\}, \\
s'''_3 &= \mathsf{lookup}_{\tau[res\mapsto\mathtt{figure}]}^{\mathtt{def},\mathtt{circle_def}}(s_2) = \bot \\
&\quad \textit{since } e''' = \{\pi \in \{\pi_2\} \mid M(\pi)(\mathtt{def}) = \mathtt{circle_def}\} = \varnothing.
\end{aligned}$$

The lookup *operations for* `figure` *and* `circle` *return $\bot$ so that, as the abstract operations over $\mathcal{ER}$ are strict on $\bot$ (Proposition 51), $s'_4 = s'_5 = s'_6 = s'''_4 = s'''_5 = s'''_6 = \bot$. 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:*

$$\begin{aligned}
s''_4 &= \mathsf{call}_{\tau[res\mapsto\mathsf{square}]}^{\mathsf{square_def}}(s''_3) = \xi_{[res\mapsto\mathsf{square}]}\left([\mathtt{this} \mapsto \{\pi_2\}] \star \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \varnothing \end{bmatrix}\right) \\
&= [\mathtt{this} \mapsto \{\pi_2\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \varnothing].
\end{aligned}$$

Since $P(\mathtt{square\_def})|_{\mathsf{out}} = [\mathtt{out} \mapsto int]$ and $\mathcal{ER}_{[\mathsf{out}\mapsto int]} = \{\bot, [\mathtt{out} \mapsto *] \star [\mathtt{next} \mapsto \varnothing,\ \mathtt{rotation} \mapsto \varnothing]\}$, 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

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

then we have

$$
\begin{aligned}
s_6'' &= \mathsf{return}^{\texttt{square\_def}}_{\tau[res\mapsto\texttt{square}]}(s_3'')(s_5'')\\
&= \mathsf{return}^{\texttt{square\_def}}_{\tau[res\mapsto\texttt{square}]}(s_2)(s_5'')\\
&= \xi_{\left[\substack{\texttt{f}\mapsto\texttt{figure},\ \texttt{n}\mapsto\texttt{figure},\\ \texttt{out}\mapsto int,\ \texttt{this}\mapsto\texttt{scan}}\right]}\left([\texttt{f}\mapsto\{\pi_2\},\texttt{n}\mapsto\varnothing,\texttt{this}\mapsto\{\overline{\pi}\}]\star\mu^\top\right)\cup\\
&\qquad\cup\left([res\mapsto *]\star[\texttt{next}\mapsto\varnothing,\ \texttt{rotation}\mapsto\varnothing]\right)\\
&= [\texttt{f}\mapsto\{\pi_2\},\texttt{n}\mapsto\varnothing,\texttt{this}\mapsto\{\overline{\pi}\}]\star\mu^\top.
\end{aligned}
$$

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 Figure 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'', you can see 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, which is not the case in the concrete state (compare σ_6'' in Example 18). 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. In this way, at the end of the method we know which creation points are reachable from the fields of such parameters. Note that, to obtain this improvement, we only need to modify the operations call and return.

Example 53. *Let us reexecute the abstract computation of Example 52, 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 nil for the formal parameter* `n`*. The abstract state* s_1 *contains now two shadow copies*

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

The same change applies to the abstract states s_2 *and* s_3'' *in Example 52. The abstract state* s_4'' *uses a new shadow copy for the actual parameter of the method* `square_def` *i.e.,*

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

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

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

Note that the abstract memory is not empty now (compare with Example 52). This is because the shadow variable $(\texttt{this})'$ *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 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 52).

There was another problem with $\mathcal{E}$, related to the fact that $\mathcal{E}$ does not distinguish between different variables (Example 35). It is not surprising that $\mathcal{ER}$ solves that problem, as shown below.

EXAMPLE 54. *Let the approximation provided by $\mathcal{ER}$ before the* $\mathtt{f} := \mathtt{new\ circle}$ *statement in Figure 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 execution 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 field* `rotation`*.*

In Example 54, the static analysis is not able to see that `f` is approximated with a *new* object, whose fields are initially set to 0 or *nil*. Since the new and put_var operations are optimal (Proposition 51), we can only improve the precision of the analysis by refining the domain. Namely, we would need a different approximation of the fields for each variable, so that the fields of *res* are approximated with $*$ or $\varnothing$ after a `new` statement.

The domain $\mathcal{ER}$ also loses precision as a consequence of the non-optimality of put_field (Proposition 51). This is illustrated by the following example.

EXAMPLE 55. *Let $\tau = \tau_{w_1}$ be as given in Example 5. Consider the execution from* $s = [\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]$ *of the statement* $\mathtt{f.next} := \mathtt{n}$ *in Example 3. It is made up of three operations from Fig-*

ure 8:

$$
\begin{aligned}
s' &= \mathsf{get\_var}^{\mathtt{f}}_{\tau}(s)\\
&= [\mathtt{f} \mapsto \{\pi_3\}, \mathit{res} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]\\
s'' &= \mathsf{get\_var}^{\mathtt{n}}_{\tau}(s)\\
&= [\mathtt{f} \mapsto \{\pi_3\}, \mathit{res} \mapsto \varnothing, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}]\\
s''' &= \mathsf{put\_field}^{\mathtt{next}}_{\tau[\mathit{res}\mapsto\mathtt{figure}],\tau[\mathit{res}\mapsto\mathtt{figure}]}(s')(s'')\\
&= \xi_\tau([\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}])\\
&= [\mathtt{f} \mapsto \{\pi_3\}, \mathtt{this} \mapsto \{\overline{\pi}\}] \star [\mathtt{next} \mapsto \{\pi_2\}, \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}].
\end{aligned}
$$

Note that the computation of s''' uses the third case in the definition of $\mathsf{put\_field}^{f}_{\tau,\tau'}$ *in Figure 8. This result is imprecise since we are putting nil inside the field* `next` *of* `f`*. Since no other field called* `next` *is reachable, a better approximation for* `next` *is $\varnothing$, which also entails that the creation points for* `rotation` *can be garbage collected.*

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 *implemented* by an element $\theta(e)$ of $\mathcal{ER}$. The idea, formalised in Definition 56, is that every variable or field must be bound in $\mathcal{ER}$ to all those creation points in e compatible with its type.

DEFINITION 56. *Let $s \subseteq \Pi$. We define $\vartheta_\tau(s) \in \mathit{Frame}^{\mathcal{ER}}_\tau$ such that, for every $v \in \mathsf{dom}(\tau)$,*

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

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

EXAMPLE 57. *Let $\tau_{w_1} = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{n} \mapsto \mathtt{figure}, \mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{scan}]$ (Example 5) and $e = \{\overline{\pi}, \pi_1, \pi_2, \pi_3\} \in \mathcal{E}_{\tau_{w_1}}$ (Example 31). Then*

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

Proposition 59 states that the implementation of Definition 56 is correct. The proof needs the following result that $\theta_\tau(e)$ is an element of $\mathcal{ER}_\tau$ and approximates exactly the same concrete states as e.

LEMMA 58. *Let $\sigma \in \Sigma_\tau$ and $e \in \mathcal{E}_\tau$. Then $\theta_\tau(e) \in \mathcal{ER}_\tau$. Moreover, $\alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e$ if and only if $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \theta_\tau(e)$.*

PROOF. We have $\theta_\tau(e) \in \mathcal{ER}_\tau$ by idempotency of ξ_τ (Proposition 47) and Definition 49.

Let $\alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e$. Let $v \in \mathsf{dom}(\tau)$. If $\tau(v) = \mathit{int}$, then $\epsilon_\tau(\sigma)(v) = * = \vartheta_\tau(e)(v)$. If $\tau(v) \in \mathcal{K}$, then every $\pi \in \epsilon_\tau(\sigma)(v)$ is such that $k(\pi) \le \tau(v)$ (Definitions 38 and 14). Moreover, $\pi = \mu(l).\pi$ for some $l \in \mathsf{rng}(\phi) \cap \mathit{Loc}$ (Definition 38). Hence $\pi \in \alpha^{\mathcal{E}}_\tau(\sigma)$ (Definition 23), and $\pi \in e$. By Definition 56 we conclude that $\pi \in \vartheta_\tau(e)(v)$. Hence $\alpha^{\mathcal{ER}}_\tau(\sigma).\phi = \epsilon_\tau(\sigma) \subseteq \vartheta_\tau(e)$. Let now $f \in \mathsf{dom}(\widetilde{\tau})$. If $\widetilde{\tau}(f) = \mathit{int}$, then $\epsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi \star \mu} \mid o \in O_\tau(\sigma)\})(f) = * = \vartheta_{\widetilde{\tau}}(e)(f)$. If $\widetilde{\tau}(f) \in \mathcal{K}$, then every $\pi \in \epsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi \star \mu} \mid o \in O_\tau(\sigma)\})(f)$ is such that $k(\pi) \le \widetilde{\tau}(f)$ (Definitions 38 and 14). Moreover, $\pi = \mu(l).\pi$ for some $l \in \mathsf{rng}(o.\phi) \cap \mathit{Loc}$ with $o \in O_\tau(\sigma)$ (Definition 38). Hence $\pi \in \alpha^{\mathcal{E}}_\tau(\sigma)$ (Definition 23), and $\pi \in e$. By Definition 56 we conclude that $\pi \in \vartheta_{\widetilde{\tau}}(e)(f)$. Hence $\alpha^{\mathcal{ER}}_{\widetilde{\tau}}(\sigma).\mu = \epsilon_{\widetilde{\tau}}(\{\widetilde{o.\phi \star \mu} \mid o \in O_\tau(\sigma)\}) \subseteq \vartheta_{\widetilde{\tau}}(e)$. In conclusion, we have $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \vartheta_\tau(e) \star \vartheta_{\widetilde{\tau}}(e)$. Since ξ_τ is monotonic (Proposition 47) and by Proposition 48, we have $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \xi_\tau(\vartheta_\tau(e) \star \vartheta_{\widetilde{\tau}}(e)) = \theta_\tau(e)$.

Conversely, let $\alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \theta_\tau(e)$. Let $\pi \in \alpha^{\mathcal{E}}_\tau(\sigma)$. By Definition 23 we have $\pi = o.\pi$ with $o \in O_\tau(\sigma)$. By Definition 42 we have $\pi \in \alpha^{\mathcal{ER}}_\tau(\sigma).\phi(v)$ for some $v \in \mathsf{dom}(\tau)$ or $\pi \in \alpha^{\mathcal{ER}}_\tau(\sigma).\mu(f)$ for some $f \in \mathsf{dom}(\widetilde{\tau})$, and hence $\pi \in \theta_\tau(e).\phi(v)$, in the first case, or $\pi \in \theta_\tau(e).\mu(f)$, in the second case. In both cases, by Definition 56 we have $\pi \in e$. Thus $\alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e$. □

We can now prove that every element of $\mathcal{E}$ represents the same set of concrete states as an element of $\mathcal{ER}$.

Proposition 59. *Let $\gamma^{\mathcal{E}}_\tau$ and $\gamma^{\mathcal{ER}}_\tau$ be the concretisation maps induced by the abstraction maps of Definitions 23 and 42, respectively. Then $\gamma^{\mathcal{E}}_\tau(\mathcal{E}_\tau) \subseteq \gamma^{\mathcal{ER}}_\tau(\mathcal{ER}_\tau)$.*

Proof. By Lemma 58, for any $e \in \mathcal{E}_\tau$, we have

$$\gamma^{\mathcal{E}}_\tau(e) = \{\sigma \in \Sigma_\tau \mid \alpha^{\mathcal{E}}_\tau(\sigma) \subseteq e\} = \{\sigma \in \Sigma_\tau \mid \alpha^{\mathcal{ER}}_\tau(\sigma) \subseteq \theta_\tau(e)\} = \gamma^{\mathcal{ER}}_\tau(\theta_\tau(e)).$$

Since this holds for all $e \in \mathcal{E}_\tau$, we have the thesis. □

The inclusion proved in Proposition 59 is strict, in general.

Example 60. *Let $\tau_{w_1} = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{n} \mapsto \mathtt{figure}, \mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{scan}]$. By Example 31 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*

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

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 49). Hence

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

6. IMPLEMENTATION

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

Note that, prior to implementing the domain $\mathcal{ER}$ as described below, we did an experimental implementation of this domain inside Loop [Spoto 2002; 2001], an analyser for programs written in a toy object-oriented language. The goal of this implementation was to investigate the feasibility of our approach. Although the performance was extremely poor, our experience with this implementation proved invaluable for the more challenging task of developing an analyser suitable for the escape analysis of real Java applications using the domain $\mathcal{ER}$.

6.1 Analysing Java Bytecode

We implemented the abstract domain $\mathcal{ER}$ inside Julia [Spoto 2004]. 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 is provided. For instance, Julia can perform *rapid type analysis* (a kind of *class analysis* [Bacon and Sweeney 1996]) 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;
(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.

We solved Problem (1) by rewording our notion of frame (Definition 36) 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 [Lindholm and Yellin 1999].

We dealt with Problem (2) by analysing some library classes only, and making *worst-case assumptions* [Cousot and Cousot 2002] 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` [Arnold et al. 2000]. 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.

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 [Dean et al. 1995]. Java bytecode subroutines (*i.e.,* the `jsr`/`ret` mechanism) is 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 [Lindholm and Yellin 1999]. The resulting graph is the same as that of *dominators* that are defined in [Aho et al. 1986] 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 [Jacobs and Poll 2003]. 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 [Spoto 2001], but composition with exception handlers uses the exceptional final state.

The abstract domain $\mathcal{ER}$ is implemented inside Julia as a Java class extending an abstract (in the sense of Java [Arnold et al. 2000]) 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 Figure 8 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 [Dovier et al. 2000] between its input and output variables. For instance, the denotation for the $\mathsf{get\_var}^v$ operation in Figure 8 is implemented as a constraint saying that the set of creation points for *res* in the output is equal to the set of creation points for v in the input. We use default reasoning to state that the other variables are unchanged. This is a generalisation of the technique we introduced in [Hill and Spoto 2003]. 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 [Bryant 1986] 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 $\mathsf{put\_var}^v$ bytecode in Figure 8 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 for the representation has, however, some little 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 $\mathsf{put\_field}$, see Figure 8). Hence conservative approximations must be made. For instance, the third case of the definition of $\mathsf{put\_field}$ 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 $\mathsf{put\_field}$ deals with the pathological case when a given $\mathsf{put\_field}$ is *always* applied to a *nil* receiver; and the second alternative deals with the case when a given $\mathsf{put\_field}$ 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 47) 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 Figure 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 its parameter (*i.e.,* the set E of Subsection 1.2). Therefore, we need information related to some internal program points. This can be obtained by placing a *watchpoint* at the beginning of each method such as `rotate` and at every exit point of the same method. 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 Figure 1 fed with a list of `Circles`; `LimVect` is a small Java program used in [Blanchet 2003]; `Dhrystone` version 2.1 is a testbench for numerical computations (most of the arrays it creates can be stack allocated); `ImageViewer` 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 $0.10j$ and `Javacc` version 3.2 are compilers' compilers; `Julia` version 0.39 is our JULIA analyser itself; `Jess` version $6.1p7$ 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 [Blanchet 2003]; 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 [Blanchet 2003].

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 of the analysis (in seconds). We first show the *static* precision of the analyses (Subsection 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 (Subsection 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 or rather heap allocated at run-time. Finally (Subsection 6.2.3) we briefly discuss the cost of the analyses.

benchmark	classes	meth.	bytec.	time	stack allocations		time per 1000 bytec.	number of constraints	linearity
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
ImageViewer	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. 9. Flow insensitive escape analyses with $\mathcal{ER}$. Fields are merged. Only `java.lang.Object` is included in the analysis.

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, the fields are approximated into one variable and, except for `java.lang.Object`, library classes are not included. As we said in Subsection 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 Figure 9, where for each benchmark we report the number of classes, methods and bytecodes analysed and the time of the analysis (in seconds).

Figure 9 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 ratio between the number of nodes in the graph of constraints and the number of strongly-connected components in this graph. 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, since fewer fixpoint iterations are needed.

Although the analyses in Figure 9 are relatively fast, you can see 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 Figure 10. They are more precise than those in Figure 9, but the analyses are also more expensive. If we also analyse some library classes, the precision of the analyses improves further. Hence 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

benchmark	classes	meth.	bytec.	time	stack allocations		time per 1000 bytec.	number of constraints	linearity
`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
`ImageViewer`	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. 10. Flow sensitive (on the operand stack only) escape analyses with $\mathcal{ER}$. Fields are merged. Only `java.lang.Object` is included in the analysis.

benchmark	classes	meth.	bytec.	time	stack allocations		time per 1000 bytec.	number of constraints	linearity
`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
`ImageViewer`	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. 11. Flow sensitive (on the operand stack only) escape analyses with the abstract domain $\mathcal{ER}$. Fields are merged. The library classes `java.lang.`{`Object`,`Character`, `String*`,`Integer`,`Number`, `CharSequence`,`AbstractStringBuilder`} and `java.util.`{`Vector`,`AbstractList`, `HashMap`, `AbstractCollection`,`Hashtable`, `AbstractMap`} are included in the analysis. For `Julia`, we also included the `bcel` libraries for bytecode manipulation.

Figure 11. We only count the creation points inside the classes of the application, so that numbers are comparable with those in Figures 9 and 10. *Wrt.* Figure 10, 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 Figure 12. The precision is just slightly better than in Figure 11, and the analyses require less time. They are sometimes even faster than those in Figure 10. 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 Figures 11 and 12). A similar behaviour has been experienced in [Rountev et al. 2001].

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

benchmark	classes	meth.	bytec.	time	stack allocations		time per 1000 bytec.	number of constraints	linearity
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
ImageViewer	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. 12. Flow sensitive (on the operand stack only) escape analyses with the abstract domain $\mathcal{ER}$. A specific approximation is used for each field. The library classes `java.lang.{Object,Character, String*,Integer, CharSequence,Number,AbstractStringBuilder}` and `java.util.{Vector,AbstractList, AbstractCollection,Hashtable, HashMap,AbstractMap}` are included in the analysis. For `Julia`, we also included the `bcel` libraries for bytecode manipulation.

very similar to those in Figure 12, confirming the claim in [Blanchet 2003] that most of the stack allocatable objects are arrays, `java.lang.StringBuffer`s 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 [Choi et al. 2003] 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 [Choi et al. 2003] 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 [Blanchet 2003]. The results reported in Figure 12 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 [Blanchet 2003], are not available anymore. See, however, Section 7 for a 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 a 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 Subsection 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

benchmark	objects in the stack		objects in the heap		memory in the stack		memory in the heap	
`Figures`	101	97.11%	3	2.89%	2024	98.82%	24	1.18%
`LimVect`	1	33.33%	2	66.66%	12	30.00%	28	70.00%
`Dhrystone`	200009	99.99%	4	0.01%	1600140	96.03%	66144	3.97%
`JLex`	672	8.92%	6854	91.08%	147060	47.77%	160772	52.23%
`JavaCup`	141875	62.37%	85564	37.63%	1580336	48.76%	1660516	51.24%
`Julia`	5534	6.56%	78748	93.44%	70764	5.57%	1198120	94.47%
`Jess`	924	7.81%	10897	92.19%	13328	4.90%	258128	95.10%
`Javacc`	26461	43.72%	34050	56.28%	604244	40.72%	883224	59.28%

Fig. 13. 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 Figure 12. `Javacc` is applied to the grammar `Java1.1.jj`. `Jess` is applied to the solution of `fullmab.clp`.

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 Figure 12 only, which are the most precise escape analyses which we manage to implement with $\mathcal{ER}$. Some results are shown in Figure 13. 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 12 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. An important observation from the same Figure 12 is that there is no exponential blow-up of the analysis time with the size of the benchmark (see the *time per* 1000 *bytecodes* column). Moreover, 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 Figure 12, which have comparable size (*i.e.,* number of bytecodes) but quite different linearity.

7. DISCUSSION

The first escape analysis [Park and Goldberg 1992] was 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 [Deutsch 1997] and finally extended to some imperative constructs and applied to very large programs [Blanchet 1998].

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 [Ruf 2000; Salcianu and

Rinard 2001; Whaley and Lam 2004]) 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 [Ruggieri and Murtagh 1988] where a *lifetime analysis* propagates the *sources* of data structures. This same idea has been used in [Agrawal 1999] where the traditional reaching definition analysis is extended to object-oriented programs. Note that, to improve efficiency, [Agrawal 1999] 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 [Whaley and Rinard 1999; Salcianu and Rinard 2001; Vivien and Rinard 2001; Choi et al. 2003; Whaley and Lam 2004]. 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 source code* 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 [Whaley and Rinard 1999; Salcianu and Rinard 2001; Vivien and Rinard 2001; Choi et al. 2003; Whaley and Lam 2004]. 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 [Vivien and Rinard 2001] have been made incremental.

A large group of escape analyses use constraints (typically, set-constraints) for expressing escape analyses. These include [Bogda and Hölzle 1999; Gay and Steensgaard 2000; Ruf 2000; Streckenbach and Snelting 2000; Rountev et al. 2001], although [Streckenbach and Snelting 2000] assumes that points-to information is available at the time of the analysis. The escape analysis in [Blanchet 2003] 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 [Choi et al. 2003] where it is said (without formal proof) that their analysis is *inherently more precise* than that defined in [Blanchet 2003]. The simplicity of Blanchet's escape analysis is however appealing, and the experimental times reported in [Blanchet 2003] currently score as the fastest of all the cited analyses that provide timings.

The way we deal with exceptions (Subsection 6.1) is largely inspired by [Jacobs and Poll 2003]. It is also similar to the technique in [Choi et al. 1999], 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 Figure 12.

Some escape analyses have been formally proved correct. Namely, [Whaley and Rinard 1999] has been proved correct in [Salcianu 2001], and [Choi et al. 2003] in [Choi et al. 2002]. Neither proof is based on abstract interpretation, and no optimality result is proved. The proof in [Blanchet 2003] 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, and no optimality result for the abstract operations is provided.

To the best of our knowledge, our notion of abstract garbage collector (Definitions 26 and 45) and its use for deriving Galois *insertions* rather than *connections* (Propositions 32 and 50) is new.

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 [Whaley and Rinard 1999; Salcianu and Rinard 2001; Vivien and Rinard 2001; Choi et al. 2003; Whaley and Lam 2004].

—Precision is also significantly affected by the call-graph construction algorithm used for the analysis which is largely independent from the analysis itself [Tip and Palsberg 2000; Grove and Chambers 2001]. However, some analyses, including our own, compute the call-graph by using the abstract information provided by the escape analysis itself. This is the case of analyses computing some form of points-to information or of our domain $\mathcal{ER}$, which approximates each variable with the set of its creation points, so that class information can be recovered.

—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 [Choi et al. 2003], since the operand stack is a distinguished feature of Java bytecode not present in Java (the target language of [Choi et al. 2003]). 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 [Whaley and Rinard 1999; Vivien and Rinard 2001; Blanchet 2003]. This idea is taken further in [Whaley and Lam 2004], 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 an escape analysis.

—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 [Blanchet 2003] 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 [Ruf 2000] is probably the most precise one *w.r.t.* synchronisation elimination. This is because it allows one to remove unnecessary synchonisation on objects which do escape from their creating thread provided, at run-time, they are locked by at most one thread. As another example the analysis in [Salcianu and Rinard 2001] 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 because 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 [Vivien and Rinard 2001]), sometimes only run-time (*dynamic*) statistics (such as [Blanchet 2003]), sometimes both (such as [Choi et al. 2003] and ourselves). Still, some statistics include the library classes (such as [Bogda and Hölzle 1999; Ruf 2000; Blanchet 2003]), others only report numbers for the user classes (such as [Choi et al. 2003], which analyses library code during the analysis but does not transform it for stack allocation, exactly as we do in Subsection 6.2). Furthermore, there is no standard set of benchmarks evaluated across all different escape analyses (the SPEC JVM98 suite is not public domain anymore, so many, such as ourselves, do not analyse it). 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 Figure 13, trying to make it as similar to that in [Blanchet 2003] as possible. However, others do not specify this information (such as [Choi et al. 2003]). Finally, analysis times are not always disclosed, making a fair comparison harder. Let us anyway compare the benchmarks that we share with [Blanchet 2003], [Choi et al. 2003] and [Rountev et al. 2001]. From Figure 13, we see that we perform equally on `Dhrystone` *w.r.t.* [Blanchet 2003] (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 [Blanchet 2003] stack allocates 26%; we stack allocate 8.92% of the run-time objects, while [Rountev et al. 2001] stack allocates 31.6%; we found 23% of the allocation sites to be stack allocatable (Figure 12) while [Rountev et al. 2001] found 27%. For `JavaCup`, we stack allocate 48.76% of the memory, while [Choi et al. 2003] stack allocates 17%. We stack allocate 39% of the allocation sites, while [Rountev et al. 2001] stack allocates 30%. For `Jess`, we only stack allocate 4.90% of the memory, while [Blanchet 2003] manages to stack allocate 27% and [Rountev et al. 2001] 17.9%. This case seems to be a consequence of our lack of a preliminary method 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 [Blanchet 2003] allocates 43%; we stack allocate 43.72% of the run-time objects, while [Rountev et al. 2001] stack allocates 45.8%; this is contradictory with the fact that we found 62% of the allocation sites to be stack allocatable (Figure 12) while [Rountev et al. 2001] 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 in 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 30). 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 25 and 34);

—That set is useful, sometimes, to restrict the possible targets of a virtual call *i.e., escape information contains class information* (Example 34).

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 [Palsberg and Schwartzbach 1991; Spoto and Jensen 2003] and *rapid type analysis* [Bacon and Sweeney 1996] although, while all objects stored in memory are considered in [Bacon and Sweeney 1996; Spoto and Jensen 2003; Palsberg and Schwartzbach 1991], only those actually reachable from the variables in scope are considered by the domains $\mathcal{E}$ and $\mathcal{ER}$ (Definitions 23 and 42). The ability to describe only the reachable objects, through the use of an abstract garbage collector (δ in Figure 7 and ξ in Figure 8), 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.

9. PROOFS

9.1 Proof of Propositions 29 and 33 in Section 4

To prove Proposition 29, we need some preliminary definitions and results. We start by defining, for every $i \in \mathbb{N}$, a map α^i_τ which, for sufficiently large i, coincides with $\alpha^{\mathcal{E}}_\tau$ (Definition 23).

Definition 61. *Let $i \in \mathbb{N}$. We define the map $\alpha^i_\tau : \wp(\Sigma_\tau) \mapsto \Pi$ as $\alpha^i_\tau(S) = \{o.\pi \mid \sigma \in S \text{ and } o \in O^i_\tau(\sigma)\}$ (see Definition 21 for O^i_τ).*

Corollary 62. *Let $S \subseteq \Sigma_\tau$ and $i \geq 0$. We have*

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

Proof. By Definitions 61 and 21. $\square$

Lemma 63 states that α_τ^i (and hence also $\alpha_\tau^{\mathcal{E}}$ itself) yields sets of creation points that do not contain garbage.

Lemma 63. *Let $\sigma \in \Sigma_\tau$ and $i \in \mathbb{N}$. Then $\alpha_\tau^i(\sigma) = \delta_\tau^i \alpha_\tau^i(\sigma)$.*

Proof. By reductivity (Proposition 28), we have $\alpha_\tau^i(\sigma) \supseteq \delta_\tau^i \alpha_\tau^i(\sigma)$. It remains to prove $\alpha_\tau^i(\sigma) \subseteq \delta_\tau^i \alpha_\tau^i(\sigma)$. Let $\sigma = \phi \star \mu$. We proceed by induction on i. We have $\alpha_\tau^0(\sigma) = \varnothing = \delta_\tau^0 \alpha_\tau^0(\sigma)$. Assume that the property holds for a given $i \in \mathbb{N}$. Let $\tau' = F(o.\pi)$ and $X = \{\mu\phi(v) \mid v \in \mathsf{dom}(\phi) \text{ and } \phi(v) \in \mathit{Loc}\}$. By Corollary 62,

$$\begin{aligned} \alpha_\tau^{i+1}(\sigma) &= \cup\{\{o.\pi\} \cup \alpha_{\tau'}^i(o.\phi \star \mu) \mid o \in X\} \\ \text{(inductive hypothesis)} &= \cup\{\{o.\pi\} \cup \delta_{\tau'}^i \alpha_{\tau'}^i(o.\phi \star \mu) \mid o \in X\}\ . \end{aligned} \tag{2}$$

By Corollary 62, we have $\alpha_{\tau'}^i(o.\phi \star \mu) \subseteq \alpha_\tau^{i+1}(\sigma)$ and, by Proposition 28, (2) is contained in

$$\cup\{\{o.\pi\} \cup \delta_{\tau'}^i \alpha_\tau^{i+1}(\sigma) \mid o \in X\}\ . \tag{3}$$

Note that, given $o \in X$, we can always find $\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K}$ such that $k(o.\pi) \leq \kappa$. Indeed, for the definition of X, there exists $v \in \mathsf{dom}(\phi) = \mathsf{dom}(\tau)$ such that $\phi(v) \in \mathit{Loc}$ and $o = \mu\phi(v)$. By Definition 11, we have $\tau(v) \in \mathcal{K}$. By Definition 14, we have $k(o.\pi) = k((\mu\phi(v)).\pi) \leq \tau(v)$. Hence letting $\kappa = \tau(v)$, (3) is

$$\begin{aligned} &\cup\{\{o.\pi\} \cup \delta_{\tau'}^i \alpha_\tau^{i+1}(\sigma) \mid o \in X,\ \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ k(o.\pi) \leq \kappa\} \\ \text{(Corollary 62)} &\subseteq \cup\{\{\pi\} \cup \delta_{\tau'}^i \alpha_\tau^{i+1}(\sigma) \mid \pi \in \alpha_\tau^{i+1}(\sigma),\ \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ k(\pi) \leq \kappa\} \\ \text{(Definition 26)} &= \delta_\tau^{i+1} \alpha_\tau^{i+1}(\sigma)\ . \end{aligned}$$

Note that the last step is correct since if $\mathtt{this} \in \mathsf{dom}(\tau)$ we have $\phi(\mathtt{this}) \neq \mathit{nil}$ (Definition 16). Hence $(\mu\phi(\mathtt{this})).\pi \in \alpha_\tau^{i+1}(\sigma)$ and $k((\mu\phi(\mathtt{this})).\pi) \leq \tau(\mathtt{this})$ (Definition 13). We conclude that, if $\mathtt{this} \in \mathsf{dom}(\tau)$, then there exists $\pi \in \alpha_\tau^{i+1}(\sigma)$ such that $k(\pi) \leq \tau(\mathtt{this})$. □

Let e be a set of creation points. We now define frames and memories which use all possible creation points in e allowed by the type environment of the variables. In this sense, they are the *richest* frames and memories containing creation points from e only.

Definition 64. *Let $\{\pi_1, \ldots, \pi_n\}$ be an enumeration without repetitions of Π. Let $l_1, \ldots, l_n$ be distinct locations. Let $e \subseteq \Pi$ and $w \in \mathsf{dom}(\tau)$ such that $\tau(w) \in \mathcal{K}$. We define*

$$L_\tau(e, w) = \{l_i \mid 1 \leq i \leq n,\ \pi_i \in e \text{ and } k(\pi_i) \leq \tau(w)\}\ ,$$

$$\overline{\phi}_\tau(e) = \left\{ \phi \in \mathit{Frame}_\tau \;\middle|\; \begin{array}{l} \text{for every } v \in \mathsf{dom}(\tau) \\ \ \text{if } \tau(v) = \mathit{int} \text{ then } \phi(v) = 0 \\ \ \text{if } \tau(v) \in \mathcal{K},\ L_\tau(e, v) = \varnothing \text{ then } \phi(v) = \mathit{nil} \\ \ \text{if } \tau(v) \in \mathcal{K},\ L_\tau(e, v) \neq \varnothing \text{ then } \phi(v) \in L_\tau(e, v) \end{array} \right\}\ ,$$

$$\overline{\mu}(e) = \left\{ \mu \in \mathit{Memory} \;\middle|\; \begin{array}{l} \mu = [l_1 \mapsto \pi_1 \star \phi_1, \ldots, l_n \mapsto \pi_n \star \phi_n] \\ \text{and } \phi_i \in \overline{\phi}_{F(\pi_i)}(e) \text{ for } i = 1, \ldots, n \end{array} \right\}\ .$$

We prove now some properties of the frames and memories of Definition 64.

Lemma 65. *Let $e_1, e_2 \subseteq \Pi$, $\phi \in \overline{\phi}_\tau(e_1)$ and $\mu \in \overline{\mu}(e_2)$. Then* *i)* $\phi\star\mu : \tau$;

ii) $\phi\star\mu \in \Sigma_\tau$ *iff* $\mathtt{this} \notin \mathsf{dom}(\tau)$ *or there exists* $\pi \in e_1$ *s.t.* $k(\pi) \le \tau(\mathtt{this})$;

iii) If $\phi\star\mu \in \Sigma_\tau$ *then* $\alpha_\tau(\phi\star\mu) \subseteq e_1 \cup e_2$.

Proof.

i) Condition 1 of Definition 14 is satisfied since $\mathsf{rng}(\phi) \cap Loc \subseteq \{l_1, \ldots, l_n\} = \mathsf{dom}(\mu)$. Moreover, if $v \in \mathsf{dom}(\phi)$ and $\phi(v) \in Loc$ then $\phi(v) \in L_\tau(e_1, v)$. Thus there exists $1 \le i \le n$ such that $\phi(v) = l_i$, $(\mu\phi(v)).\pi = \pi_i$ and $k((\mu\phi(v)).\pi) = k(\pi_i) \le \tau(v)$. Condition 2 of Definition 14 holds because if $o \in \mathsf{rng}(\mu)$ then $o.\phi = \phi_i$ for some $1 \le i \le n$. Since $\phi_i \in \overline{\phi}_{F(\pi_i)}(e)$, reasoning as above we conclude that ϕ_i is $F(\pi_i)$-correct *w.r.t.* μ. Then $\phi\star\mu : \tau$.

ii) By point i, we know that $\phi\star\mu : \tau$. From Definition 16, we have $\phi\star\mu \in \Sigma_\tau$ if and only if $\mathtt{this} \notin \mathsf{dom}(\tau)$ or $\phi(\mathtt{this}) \neq nil$. By Definition 64, the latter case holds if and only if $L_\tau(e_1, \mathtt{this}) \neq \varnothing$ *i.e.,* if and only if there exists $\pi \in e_1$ such that $k(\pi) \le \tau(\mathtt{this})$.

iii) Since $\phi\star\mu \in \Sigma_\tau$, the α_τ map is defined (Definition 23). Let

$$L = (\mathsf{rng}(\phi) \cup (\cup\{\mathsf{rng}(o.\phi) \mid o \in \mathsf{rng}(\mu)\})) \cap Loc\ .$$

Since $\phi \in \overline{\phi}_\tau(e_1)$ and $o.\phi \in \overline{\phi}_{F(o.\pi)}(e_2)$ for every $o \in \mathsf{rng}(\mu)$, by Definition 64, we have

$$\{\mu(l).\pi \mid l \in L\} \subseteq e_1 \cup e_2\ .$$

By Definition 23, we conclude that

$$\alpha_\tau(\phi\star\mu) \subseteq \{\mu(l).\pi \mid l \in L\} \subseteq e_1 \cup e_2\ .$$

□

Lemma 66 gives an explicit definition of the abstraction of the set of states constructed from the frames and memories of Definition 64.

Lemma 66. *Let* $e_1, e_2 \subseteq \Pi$, $j \in \mathbb{N}$ *and*

$$A^j = \alpha_\tau^{j+1}(\{\phi\star\mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e_1) \text{ and } \mu \in \overline{\mu}(e_2)\})\ .$$

Then

$$A^j = \begin{cases} \varnothing & \textit{if } \mathtt{this} \in \mathsf{dom}(\tau) \textit{ and there is no } \pi \in e_1 \textit{ s.t. } k(\pi) \le \tau(\mathtt{this}) \\ \cup\left\{ \{\pi\} \cup \delta^j_{F(\pi)}(e_2) \;\middle|\; \begin{array}{l} v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ \pi \in e_1,\ k(\pi) \le \tau(v) \end{array} \right\} & \textit{otherwise.} \end{cases}$$

Proof. We proceed by induction over j. By Lemma 65.ii, if $j = 0$ we have

$$A^0 = \begin{cases} \varnothing & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and there is no } \pi \in e_1 \text{ s.t. } k(\pi) \le \tau(\mathtt{this}) \\ \left\{ o.\pi \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_\tau(e_1),\ \mu \in \overline{\mu}(e_2),\ v \in \mathsf{dom}(\phi) \\ \phi(v) \in Loc,\ o = \mu\phi(v) \end{array} \right\} & \text{otherwise.} \end{cases}$$

By Definition 64, the latter case is equal to

$$\left\{ \pi_i \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ 1 \le i \le n,\ \pi_i \in e_1 \\ k(\pi_i) \le \tau(v) \end{array} \right\} = \cup \left\{ \{\pi\} \cup \delta^0_{F(\pi)}(e_2) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau) \\ \tau(v) \in \mathcal{K},\ \pi \in e_1 \\ k(\pi) \le \tau(v) \end{array} \right\}.$$

Assume now that the result holds for a given $j \in \mathbb{N}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and there is no $\pi \in e_1$ such that $k(\pi) \le \tau(\mathtt{this})$, by Lemma 65.ii, we have $A^{j+1} = \varnothing$. Otherwise, by Corollary 62 we have

$$A^{j+1} = \cup \left\{ \{o.\pi\} \cup \alpha^{j+1}_{F(o.\pi)}(o.\phi \star \mu) \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_\tau(e_1),\ \mu \in \overline{\mu}(e_2),\ v \in \mathsf{dom}(\phi) \\ \phi(v) \in Loc,\ o = \mu\phi(v) \end{array} \right\}. \tag{4}$$

As for the base case, we know that $o.\pi$ ranges over $\{\pi \in e_1 \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K},\ k(\pi) \le \tau(v)\}$. Since $o.\phi \in \overline{\phi}_{F(o.\pi)}(e_2)$ is arbitrary (Definition 64), by the inductive hypothesis, (4) becomes

$$\begin{aligned} &\cup \left\{ \{\pi\} \cup \alpha^{j+1}_{F(\pi)} \left(\left\{ \phi \star \mu \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_{F(\pi)}(e_2) \\ \mu \in \overline{\mu}(e_2) \end{array} \right\} \right) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ \pi \in e_1,\ k(\pi) \le \tau(v) \end{array} \right\} \\ &= \cup \left\{ \{\pi\} \cup \delta^{j+1}_{F(\pi)}(e_2) \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K},\ \pi \in e_1,\ k(\pi) \le \tau(v) \right\}. \end{aligned}$$

□

Corollary 67. *Let* $e_1, e_2 \subseteq \Pi$. *Let*

$$A_\tau(e_1, e_2) = \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e_1) \text{ and } \mu \in \overline{\mu}(e_2)\}).$$

Then

i) $$A_\tau(e_1, e_2) = \begin{cases} \varnothing & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and no } \pi \in e_1 \text{ is s.t. } k(\pi) \le \tau(\mathtt{this}) \\ \cup \left\{ \{\pi\} \cup \delta_{F(\pi)}(e_2) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\phi),\ \tau(v) \in \mathcal{K} \\ \pi \in e_1,\ k(\pi) \le \tau(v) \end{array} \right\} & \text{otherwise,} \end{cases}$$

ii) $A_\tau(e_1, e_1) = \delta_\tau(e_1)$.

Proof. Point i follows by Lemma 66 since j is arbitrary. Point ii follows from point i and Definition 26. □

Corollary 68. *Let* $\kappa \in \mathcal{K}$, $\tau = [res \mapsto \kappa]$, p *be a predicate over* Π *and* $e \subseteq \Pi$ *be such that there exists* $\pi \in e$ *such that* $k(\pi) \le \tau(res)$ *and* $p(\pi)$ *holds. Then*

$$\begin{aligned} &\alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ p(\mu\phi(res).\pi)\}) \\ &\qquad = \cup\{\{\pi\} \cup \delta_{F(\pi)}(e) \mid \pi \in e,\ k(\pi) \le \tau(res),\ p(\pi)\}. \end{aligned}$$

Proof. Let $j \in \mathbb{N}$. By the hypothesis on e and Corollary 62 we have

$$\begin{aligned} &\alpha^{j+1}_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ p(\mu\phi(res).\pi)\}) \\ &= \cup\{\{o.\pi\} \cup \alpha^j_{F(o.\pi)}(o.\phi \star \mu) \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ o = \mu\phi(res),\ p(o.\pi)\} \\ &= \cup\{\{\pi\} \cup \alpha^j_{F(\pi)}(\phi' \star \mu) \mid \pi \in e,\ k(\pi) \le \tau(res),\ p(\pi),\ \phi' \in \overline{\phi}_{F(\pi)}(e),\ \mu \in \overline{\mu}(e)\}. \end{aligned}$$

Since j is arbitrary we have

$$\begin{aligned}&\alpha_\tau(\{\phi\star\mu\in\Sigma_\tau \mid \phi\in\overline{\phi}_\tau(e),\ \mu\in\overline{\mu}(e),\ p(\mu\phi(res).\pi)\})\\&=\cup\left\{\{\pi\}\cup\alpha_{F(\pi)}\left(\left\{\phi'\star\mu \,\middle|\, \begin{array}{l}\phi'\in\overline{\phi}_{F(\pi)}(e)\\ \mu\in\overline{\mu}(e)\end{array}\right\}\right) \,\middle|\, \begin{array}{l}\pi\in e,\ p(\pi)\\ k(\pi)\le\tau(res)\end{array}\right\},\end{aligned}$$

and the thesis follows by Corollary 67.ii. □

We now prove Proposition 29. To do this, we will use the set of states constructed from the frames and memories in Definition 64 to show that $\alpha_\tau^{\mathcal{E}}$ is onto.

Proof *of Proposition 29 at page 20*

∘ We first prove that $\mathsf{fp}(\delta_\tau)=\mathsf{rng}(\alpha_\tau)$. Let $X\subseteq\Sigma_\tau$ and $i\in\mathbb{N}$. By Lemma 63 and monotonicity (Proposition 28) we have

$$\alpha_\tau^i(X)=\cup\{\alpha_\tau^i(\sigma)\mid\sigma\in X\}=\cup\{\delta_\tau^i\alpha_\tau^i(\sigma)\mid\sigma\in X\}\subseteq\delta_\tau^i\alpha_\tau^i(X)\subseteq\delta_\tau\alpha_\tau^i(X)\ .$$

The converse inclusion $\alpha_\tau^i(X)\subseteq\delta_\tau\alpha_\tau^i(X)$ holds because δ_τ is reductive (Proposition 28). Then $\alpha_\tau^i(X)\in\mathsf{fp}(\delta_\tau)$. Since i is arbitrary we have $\alpha_\tau(X)\in\mathsf{fp}(\delta_\tau)$. Conversely, let $e\in\mathsf{fp}(\delta_\tau)$ and

$$X=\{\phi\star\mu\in\Sigma_\tau\mid\phi\in\overline{\phi}_\tau(e),\ \mu\in\overline{\mu}(e)\}\ .$$

By Corollary 67.ii and since $e\in\mathsf{fp}(\delta_\tau)$, we have $\alpha_\tau(X)=\delta_\tau(e)=e$.

Since δ_τ is reductive (Proposition 28), we have $\varnothing=\delta_\tau(\varnothing)$ *i.e.*, $\varnothing\in\mathsf{fp}(\delta_\tau)$.

If `this` $\in\mathsf{dom}(\tau)$, every $\sigma\in\Sigma_\tau$ is such that $\alpha_\tau(\sigma)\neq\varnothing$, since `this` cannot be unbound (Definition 16). Then $\alpha_\tau(X)=\varnothing$ if and only if $X=\varnothing$. □

The proof of Proposition 33 requires some preliminary results.

Corollary 69 states that if we know that the approximation of a set of concrete states S is some $e\subseteq\Pi$, then we can conclude that a better approximation of S is $\delta(e)$. In other words, garbage is never used in the approximation.

Corollary 69. *Let $S\subseteq\Sigma_\tau$ and $e\subseteq\Pi$. Then $\alpha_\tau(S)\subseteq\delta_\tau(e)$ if and only if $\alpha_\tau(S)\subseteq e$.*

Proof. Assume that $\alpha_\tau(S)\subseteq\delta_\tau(e)$. By reductivity (Proposition 28) we have $\alpha_\tau(S)\subseteq e$. Conversely, assume that $\alpha_\tau(S)\subseteq e$. By Proposition 29 and monotonicity (Proposition 28) we have $\alpha_\tau(S)=\delta_\tau\alpha_\tau(S)\subseteq\delta_\tau(e)$. □

Lemma 70 states that integer values, *nil* and the name of the variables are not relevant to the definition of α (Definition 23).

Lemma 70. *Let $\phi'\star\mu\in\Sigma_{\tau'}$ and $\phi''\star\mu\in\Sigma_{\tau''}$ such that $\mathsf{rng}(\phi')\cap Loc=\mathsf{rng}(\phi'')\cap Loc$. Then $\alpha_{\tau'}(\phi'\star\mu)=\alpha_{\tau''}(\phi''\star\mu)$.*

Proof. From Definition 23. □

Lemma 71 says that if we consider all the concrete states approximated by some $e\subseteq\Pi$ and we restrict their frames, then the resulting set of states is approximated by $\delta(e)$. In other words, the operation δ garbage collects all objects that, because of the restriction, are not longer reachable.

LEMMA 71. *Let* $vs \subseteq \mathsf{dom}(\tau)$. *Then*

$$\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \textit{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) = \delta_{\tau|_{-vs}}(e) \ .$$

PROOF. We have

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) \\ &= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) \ , \end{aligned} \tag{5}$$

since if $\phi \star \mu \in \Sigma_\tau$ then $\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}}$. We have that if $\alpha_\tau(\phi \star \mu) \subseteq e$ then $\alpha_{\tau|_{-vs}}(\phi|_{-vs} \star \mu) \subseteq e$. Hence (5) is contained in e. By Corollary 69, (5) is also contained in $\delta_{\tau|_{-vs}}(e)$. But also the converse inclusion holds, since in (5) we can restrict the choice of $\phi \star \mu \in \Sigma_\tau$, so that (5) contains

$$\alpha_{\tau|_{-vs}}\left(\left\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \,\middle|\, \begin{array}{l} \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq e \\ \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e) \end{array}\right\}\right) \ . \tag{6}$$

By points ii and iii of Lemma 65, (6) is equal to

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\}) \\ (\text{Definition 64}) &= \alpha_{\tau|_{-vs}}(\{\phi \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{\phi}_{\tau|_{-vs}}(e) \text{ and } \mu \in \overline{\mu}(e)\}) \\ (\text{Corollary 67.ii}) &= \delta_{\tau|_{-vs}}(e) \ . \end{aligned}$$

□

We are now ready to prove the correctness and optimality of the abstract operations in Figure 7.

Proof *of Proposition 33 at page 21*

∘ By the theory of abstract interpretation [Cousot and Cousot 1977], given $e \in \mathcal{E}_\tau$, the concretisation map induced by the abstraction map of Definition 23 is

$$\gamma_\tau(e) = \{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\} \ .$$

Moreover, the optimal abstract counterpart of a concrete operation *op* is $\alpha op \gamma$.

We consider every operation in Figure 6 and we compute the induced optimal abstract operation, which will always coincide with that reported in Figure 7.

Note that all the operations in Figure 6 use states in Σ_τ with `this` $\in \mathsf{dom}(\tau)$ (Figure 5). By Proposition 29 we have $\gamma_\tau(\varnothing) = \varnothing$. Then the powerset extension of the operations in Figure 6 are strict on $\varnothing$. The only exception is the second argument of `return`, which is a state whose frame is not required to contain `this` (Figure 5). The operation $\cup$ is not the powerset extension of an operation in Figure 6. Then it is not strict in general. Hence, in the following, we will consider just the cases when the arguments of the abstract counterparts of the operations in Figure 6 are not $\varnothing$ (except for the second argument of `return` and for $\cup$).

In this proof, we will use the following properties.

P1 If $e \in \mathcal{E}_\tau$, $e \neq \varnothing$ and `this` $\in \mathsf{dom}(\tau)$ then there exists $\pi \in e$ such that $k(\pi) \leq \tau(\texttt{this})$.

P2 If $e \in \mathcal{E}_\tau$, $e \neq \varnothing$ and `this` $\in \mathsf{dom}(\tau)$ then there exists $\sigma \in \Sigma_\tau$ such that $\alpha_\tau(\sigma) \subseteq e$.

P3 $\alpha_\tau\gamma_\tau$ is the identity map.

P1 holds since $e = \delta_\tau(e)$ (Definition 30) so that by Definition 26, we can conclude that there exists such a π. To see that P2 is a consequence of P1, let π be as defined in P1; then, letting $\sigma = [\texttt{this} \mapsto l] \star [l \mapsto \pi \star \Im(F(\pi))]$ for some $l \in Loc$, we have $\sigma \in \Sigma_\tau$. Moreover, by Definition 23, $\alpha_\tau(\sigma) = \{\pi\} \subseteq e$ so that P2 holds. By Proposition 32, α_τ is a Galois insertion and hence, P3 holds.

<u>nop</u>
By P3 we have

$$\alpha_\tau(\mathsf{nop}_\tau(\gamma_\tau(e))) = \alpha_\tau\gamma_\tau(e) = e\ .$$

<u>get_int, get_nil, get_var</u>

$$\begin{aligned}
&\alpha_{\tau[res\mapsto int]}(\mathsf{get_int}^i_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau[res\mapsto int]}(\{\phi[res \mapsto i] \star \mu \mid \phi \star \mu \in \gamma_\tau(e)\})\\
(*) &= \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \star \mu \in \gamma_\tau(e)\}) = \alpha_\tau\gamma_\tau(e) = e\ ,
\end{aligned}$$

where point $*$ follows by Lemma 70 since $res \notin \mathsf{dom}(\tau)$. For the same reason, point $*$ follows if res is bound to nil or to some $\phi(v)$ with $v \in \mathsf{dom}(\tau)$. Thus the proof above is also a proof of the optimality of get_nil and of get_var.

<u>expand</u>

$$\begin{aligned}
&\alpha_{\tau[v\mapsto t]}(\mathsf{expand}^{v:t}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau[v\mapsto t]}(\{\phi[v \mapsto \Im(t)] \star \mu \mid \phi \star \mu \in \gamma_\tau(e)\})\\
(*) &= \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \star \mu \in \gamma_\tau(e)\}) = \alpha_\tau\gamma_\tau(e) = e\ ,
\end{aligned}$$

where point $*$ follows by Lemma 70, since $\Im(t) \in \{0, nil\}$ and $v \notin \mathsf{dom}(\tau)$.

<u>restrict</u>

$$\begin{aligned}
&\alpha_{\tau|_{-vs}}(\mathsf{restrict}^{vs}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau|_{-vs}}(\mathsf{restrict}^{vs}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\})\\
\text{(Lemma 71)} &= \delta_{\tau|_{-vs}}(e)\ .
\end{aligned}$$

<u>is_nil</u>

$$\begin{aligned}
&\alpha_{\tau[res\mapsto int]}(\mathsf{is_nil}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau[res\mapsto int]}(\mathsf{is_nil}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{\tau[res\mapsto int]}(\{\phi[res \mapsto 1] \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\})\\
\text{(Lemma 70)} &= \alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\})\\
\text{(Lemma 71)} &= \delta_{\tau|_{-res}}(e)\\
\text{(Definition 26)} &= \delta_{\tau[res\mapsto int]}(e)\ .
\end{aligned}$$

put_var

$$
\begin{aligned}
&\alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{\tau|_{-res}}(\{\phi[v \mapsto \phi(res)]|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) \ . \qquad (7)
\end{aligned}
$$

Observe that $\mathsf{rng}(\phi[v \mapsto \phi(res)]|_{-res}) = \mathsf{rng}(\phi|_{-v})$ so that, by Lemmas 70 and 71, (7) is equal to

$$
\alpha_{\tau|_{-v}}(\{\phi|_{-v} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq e\}) = \delta_{\tau|_{-v}}(e) \ .
$$

call

$$
\begin{aligned}
&\alpha_{P(\nu)|_{-\mathsf{out}}}(\mathsf{call}_\tau^{\nu,v_1,\ldots,v_n}(\gamma_\tau(e)))\\
&= \alpha_{P(\nu)|_{-\mathsf{out}}}(\mathsf{call}_\tau^{\nu,v_1,\ldots,v_n}(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{P(\nu)|_{-\mathsf{out}}}\left(\left\{\begin{bmatrix} \iota_1 \mapsto \phi(v_1), \ldots, \iota_n \mapsto \phi(v_n), \\ \mathtt{this} \mapsto \phi(res) \end{bmatrix} \star \mu \,\middle|\, \begin{matrix} \phi \star \mu \in \Sigma_\tau \text{ and} \\ \alpha_\tau(\phi \star \mu) \subseteq e \end{matrix}\right\}\right)\\
(*) &= \alpha_{\tau|_{\{v_1,\ldots,v_n,res\}}}(\{\phi|_{\{v_1,\ldots,v_n,res\}} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq e\})\\
(**) &= \delta_{\tau|_{\{v_1,\ldots,v_n,res\}}}(e) \ ,
\end{aligned}
$$

where point $*$ follows by Lemma 70 and point $**$ follows by Lemma 71.

is_true, is_false

$$
\begin{aligned}
&\alpha_\tau(\mathsf{is\_true}_\tau(\gamma_\tau(e)))\\
&= \alpha_\tau(\{\phi \star \mu \in \gamma_\tau(e) \mid \phi(res) \geq 0\})\\
(\text{Lemma 70}) &= \alpha_\tau \gamma_\tau(e) = e \ .
\end{aligned}
$$

The optimality of is_false follows by a similar proof.

new

Let $\kappa = k(\pi)$. Since $res \notin \mathsf{dom}(\tau)$ we have

$$
\begin{aligned}
&\alpha_{\tau[res \mapsto \kappa]}(\mathsf{new}_\tau^\pi(\gamma_\tau(e)))\\
&= \alpha_{\tau[res \mapsto \kappa]}(\mathsf{new}_\tau^\pi(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq e\}))\\
&= \alpha_{\tau[res \mapsto \kappa]}\left(\left\{\begin{matrix} \phi[res \mapsto l] \star \\ \star\, \mu[l \mapsto \pi \star \Im(F(\kappa))] \end{matrix} \,\middle|\, \begin{matrix} \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq e \\ l \in Loc \setminus \mathsf{dom}(\mu) \end{matrix}\right\}\right)\\
&= \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq e\}) \cup \qquad (8)\\
&\quad \cup\, \alpha_{[res \mapsto \kappa]}\left(\left\{\begin{matrix} [res \mapsto l] \star \\ \star [l \mapsto \pi \star \Im(F(\kappa))] \end{matrix} \,\middle|\, \begin{matrix} \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq e \\ l \in Loc \setminus \mathsf{dom}(\mu) \end{matrix}\right\}\right) . \qquad (9)
\end{aligned}
$$

We have that (8) is equal to e. By P2 and Definition 23, (9) is equal to $\{\pi\}$.

<u>=, +</u>

$$
\begin{aligned}
&\alpha_\tau(=_\tau(\gamma_\tau(e_1))(\gamma_\tau(e_2)))\\
&= \alpha_\tau(\{=_\tau(\sigma_1)(\sigma_2) \mid \sigma_1 \in \gamma_\tau(e_1),\ \sigma_2 \in \gamma_\tau(e_2)\})\\
(\mathrm{P2}) &= \alpha_\tau(\{\sigma_2 \mid \sigma_2 \in \gamma_\tau(e_2)\})\\
&= \alpha_\tau\gamma_\tau(e_2) = e_2\ .
\end{aligned}
$$

The optimality of + follows by a similar proof.

<u>return</u>
Let $\tau' = \tau[res \mapsto P(\nu)(\mathsf{out})]$, $\tau'' = P(\nu)|_{\mathsf{out}}$ and $L = \mathsf{rng}(\phi_1|_{-res}) \cap Loc$.

$$
\begin{aligned}
&\alpha_{\tau'}(\mathsf{return}^\nu_\tau(\gamma_\tau(e_1))(\gamma_{\tau''}(e_2)))\\
&= \alpha_{\tau'}(\mathsf{return}^\nu_\tau(\{\sigma_1 \in \Sigma_\tau \mid \alpha_\tau(\sigma_1) \subseteq e_1\})(\{\sigma_2 \in \Sigma_{\tau''} \mid \alpha_{\tau''}(\sigma_2) \subseteq e_2\}))\\
&= \alpha_{\tau'}\left(\left\{\phi_1|_{-res}[res \mapsto \phi_2(\mathsf{out})] \star \mu_2 \;\middle|\; \underbrace{\begin{array}{c}\phi_1 \star \mu_1 \in \Sigma_\tau,\ \phi_2 \star \mu_2 \in \Sigma_{\tau''}\\ \alpha_\tau(\phi_1 \star \mu_1) \subseteq e_1\\ \alpha_{\tau''}(\phi_2 \star \mu_2) \subseteq e_2,\ \mu_1 \lhd_L \mu_2\end{array}}_{Cond}\right\}\right)
\end{aligned}
$$

$$
(*) = \alpha_{\tau|_{-res}}(\{\phi_1|_{-res} \star \mu_2 \mid Cond\}) \cup \tag{10}
$$

$$
\cup\, \alpha_{\tau''}(\{\phi_2 \star \mu_2 \mid Cond\}) \tag{11}
$$

where point $*$ follows by Lemma 70. Since $\alpha_{\tau''}(\phi_2 \star \mu_2) \subseteq e_2$, an upper bound of (11) is e_2. But e_2 is also a lower bound of (11) since, by Lemma 65.iii, a lower bound of (11) is

$$
\alpha_{\tau''}\left(\left\{\phi_2 \star \mu_2 \;\middle|\; \begin{array}{l}\phi_1 \in \overline{\phi}_\tau(e_1),\ \mu_1 \in \overline{\mu}(e_1)\\ \phi_2 \in \overline{\phi}_{\tau''}(e_2),\ \mu_2 \in \overline{\mu}(e_2)\end{array}\right\}\right)
$$

which by Corollary 67.ii is equal to e_2. Note that the condition $\mu_1 \lhd_L \mu_2$ is satisfied by Definition 64.

Instead (10), is

$$
\cup \left\{\{o.\pi\} \cup \alpha_{F(o.\pi)}(o.\phi \star \mu_2) \;\middle|\; \begin{array}{l} v \in \mathsf{dom}(\phi_1|_{-res}),\ \phi_1|_{-res}(v) \in Loc\\ o = \mu_2\phi_1|_{-res}(v),\ Cond\end{array}\right\}
$$

which, since $\mu_1 \lhd_L \mu_2$, is equal to

$$
\begin{aligned}
&\cup \left\{\{o.\pi\} \cup \alpha_{F(o.\pi)}(o.\phi \star \mu_2) \;\middle|\; \begin{array}{l} v \in \mathsf{dom}(\phi_1|_{-res}),\ \phi_1|_{-res}(v) \in Loc\\ o = \mu_1\phi_1|_{-res}(v),\ Cond\end{array}\right\}\\
(*) &\subseteq \cup \left\{\{o.\pi\} \cup \delta_{F(o.\pi)}(\Pi) \;\middle|\; \begin{array}{l} v \in \mathsf{dom}(\phi_1|_{-res}),\ \phi_1|_{-res}(v) \in Loc\\ o = \mu_1\phi_1|_{-res}(v),\ Cond\end{array}\right\}\\
(**) &\subseteq \cup \{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K},\ \pi \in e_1,\ k(\pi) \leq \kappa,\ Cond\}\\
&\subseteq \cup \{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K},\ \pi \in e_1,\ k(\pi) \leq \kappa\}\ ,
\end{aligned} \tag{12}
$$

where point $*$ follows by Lemma 63 and point $**$ holds since $Cond$ requires that $\alpha_\tau(\phi_1 \star \mu_1) \subseteq e_1$. But (12) is also a lower bound of (10), since (10) contains

$$\alpha_{\tau|_{-res}}\left(\left\{\phi_1|_{-res} \star \mu_2 \in \Sigma_{\tau|_{-res}} \;\middle|\; \begin{array}{l}\phi_1 \in \overline{\phi}_\tau(e_1),\ \mu_1 \in \overline{\mu}(e_1),\\ \phi_2 = \Im(P(\nu)|_{\mathsf{out}}),\ \mu_2 \in \overline{\mu}(\Pi)\end{array}\right\}\right)$$
$$= \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e_1), \mu \in \overline{\mu}(\Pi)\}) ,$$

which by Corollary 67.i is equal to (12).

<u>get_field</u>

Let $\tau' = \tau[res \mapsto F(\tau(res))(f)]$ and $\tau'' = [res \mapsto F(\tau(res))(f)]$. We have

$$\begin{aligned}
&\alpha_{\tau'}(\mathsf{get_field}^f_\tau(\gamma_\tau(e)))\\
&= \alpha_{\tau'}(\mathsf{get_field}^f_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq e\}))\\
&= \alpha_{\tau'}\left(\left\{\phi|_{-res}[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \;\middle|\; \begin{array}{l}\phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq nil,\\ \alpha_\tau(\phi \star \mu) \subseteq e\end{array}\right\}\right)\\
&= \alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq nil,\ \alpha_\tau(\phi \star \mu) \subseteq e\}) \cup\\
&\quad \cup\, \alpha_{\tau''}\left(\left\{[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \;\middle|\; \begin{array}{l}\phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq nil,\\ \alpha_\tau(\phi \star \mu) \subseteq e\end{array}\right\}\right) . \qquad (13)
\end{aligned}$$

(13) is equal to $\varnothing$ if $\{\pi \in e \mid k(\pi) \leq \tau(res)\} = \varnothing$, since in such a case the condition $\phi(res) \neq nil$ cannot be satisfied. Since $\alpha_\tau(\phi \star \mu) \subseteq e$, an upper bound of (13) is e. By Corollary 69, also $\delta_{\tau'}(e)$ is an upper bound of (13). But it is also a lower bound of (13), since, from the hypothesis on e and from points ii and iii of Lemma 65, (13) contains

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\}) \cup\\
&\cup\, \alpha_{\tau''}(\{[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \in \Sigma_{\tau''} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e)\})\\
(*) = {} &\alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e),\ \mu \in \overline{\mu}(e)\}) \cup\\
&\cup\, \alpha_{\tau''}(\{\phi \star \mu \in \Sigma_{\tau''} \mid \phi \in \overline{\phi}_{\tau''}(e),\ \mu \in \overline{\mu}(e)\})\\
(**) = {} &\delta_{\tau|_{-res}}(e) \cup \delta_{\tau''}(e) = \delta_{\tau'}(e) ,
\end{aligned}$$

where point $*$ follows by Definition 64 and point $**$ follows by Corollary 67.ii.

<u>lookup</u>

$$\begin{aligned}
&\alpha_\tau(\mathsf{lookup}^{m,\nu}_\tau(\gamma_\tau(e)))\\
&= \alpha_\tau(\mathsf{lookup}^{m,\nu}_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq e\}))\\
&= \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \underbrace{\alpha_\tau(\phi \star \mu) \subseteq e,\ \phi(res) \neq nil,\ M((\mu\phi(res)).\pi)(m) = \nu}_{Cond}\}). \qquad (14)
\end{aligned}$$

Equation (14) is equal to $\varnothing$ if there is no $\pi \in e$ such that $k(\pi) \leq \tau(res)$ and $M(\pi)(m) = \nu$, because in such a case the condition $M((\mu\phi(res)).\pi)(m) = \nu$ cannot be satisfied. Otherwise, it is equal to

$$\alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ Cond\}) \cup \qquad (15)$$
$$\cup\, \alpha_{\tau|_{res}}(\{\phi|_{res} \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ Cond\}) . \qquad (16)$$

Since $Cond$ requires that $\alpha_\tau(\phi \star \mu) \subseteq e$, by Corollary 69 an upper bound of (15) is $\delta_{\tau|_{-res}}(e)$. But it is also a lower bound of (15), since a lower bound of (15) is

$$
\begin{aligned}
&\alpha_{\tau|_{-res}}\left(\left\{\phi|_{-res} \star \mu \in \Sigma_{\tau|_{-res}} \;\middle|\; \begin{array}{l} \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ \phi(res) \neq nil \\ M((\mu\phi(res)).\pi)(m) = \nu \end{array}\right\}\right) \\
(*) &= \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e),\ \mu \in \overline{\mu}(e)\}) \\
(**) &= \delta_{\tau|_{-res}}(e)\ .
\end{aligned}
$$

Point $*$ follows from the hypothesis on e. Point $**$ follows by Corollary 67.ii.

Instead, (16) is contained in

$$
\begin{aligned}
&\alpha_{\tau|_{res}}(\{\phi|_{res} \star \mu \in \Sigma_{\tau|_{res}} \mid \phi \in \overline{\phi}_\tau(e),\ \mu \in \overline{\mu}(e),\ M((\mu\phi(res)).\pi)(m) = \nu\}) \\
&\quad = \alpha_{\tau|_{res}}(\{\phi \star \mu \in \Sigma_{\tau|_{res}} \mid \phi \in \overline{\phi}_{\tau|_{res}}(e),\ \mu \in \overline{\mu}(e),\ M((\mu\phi(res)).\pi)(m) = \nu\})\ ,
\end{aligned}
$$

which, by Corollary 68, is

$$
\cup\{\{\pi\} \cup \delta_{F(\pi)}(e) \mid \pi \in e,\ k(\pi) \leq \tau(res),\ M(\pi)(m) = \nu\}\ .
$$

<u>put_field</u>

$$
\begin{aligned}
&\alpha_{\tau|_{-res}}(\mathsf{put\_field}_{\tau,\tau'}(\gamma_\tau(e_1))(\gamma_\tau(e_2))) \\
&= \alpha_{\tau|_{-res}}(\mathsf{put\_field}_{\tau,\tau'}(\{\sigma_1 \in \Sigma_\tau \mid \alpha_\tau(\sigma_1) \subseteq e_1\})(\{\sigma_2 \in \Sigma_{\tau'} \mid \alpha_{\tau'}(\sigma_2) \subseteq e_2\})) \\
&= \alpha_{\tau|_{-res}}\left(\left\{\begin{array}{c} \phi_2|_{-res} \star \mu_2[l \mapsto \mu_2(l).\pi \star \\ \star \mu_2(l).\phi[f \mapsto \phi_2(res)]] \end{array} \;\middle|\; \begin{array}{l} \phi_1 \star \mu_1 \in \Sigma_\tau,\ \phi_2 \star \mu_2 \in \Sigma_{\tau'} \\ \alpha_\tau(\phi_1 \star \mu_1) \subseteq e_1,\ \alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq e_2 \\ (l = \phi_1(res)) \neq nil,\ \mu_1 \lhd_l \mu_2 \end{array}\right\}\right)
\end{aligned}
\tag{17}
$$

which is $\varnothing$ if there is no $\pi \in e_1$ such that $k(\pi) \leq \tau(res)$, since in such a case the condition $\phi_1(res) \neq nil$ cannot be satisfied. Otherwise, note that the operation put_field copies the value of $\phi_2(res)$, which is obviously reachable from ϕ_2, inside a field. Since $\alpha_{\tau'}(\phi_2 \star \mu_2) \subseteq e_2$, we conclude that an upper bound of (17) is e_2. Then $\delta_{\tau|_{-res}}(e_2)$ is also an upper bound of (17) (Corollary 69). We show that it is also a lower bound. Let $\pi_1 \in e_1$ be such that $k(\pi_1) \leq \tau(\mathtt{this})$ (possible for P1) and $\pi_2 \in e_1$ be such that $k(\pi_2) \leq \tau(res)$ (possible for the hypothesis on e_1). Let $o_1 = \pi_1 \star \Im(F(\pi_1))$ and $o_2 = \pi_2 \star \Im(F(\pi_2))$. We obtain the following lower bound of (17) by choosing special cases for ϕ_1, μ_1, ϕ_2 and μ_2.

$$
\alpha_{\tau|_{-res}}\left(\left\{\begin{array}{c} \phi_2|_{-res} \star \mu_2[l_2 \mapsto \mu_2(l_2).\pi \star \\ \star \mu_2(l_2).\phi[f \mapsto \phi_2(res)]] \end{array} \;\middle|\; \begin{array}{l} \phi_1 = \Im(\tau)[\mathtt{this} \mapsto l_1, res \mapsto l_2] \\ \phi_2 \in \overline{\phi}_{\tau'}(e_2),\ \mu_2' \in \overline{\mu}(e_2) \\ \phi_2 \star \mu_2' \in \Sigma_{\tau'} \\ \mu_1 = \mu_2 = \mu_2'[l_1 \mapsto o_1, l_2 \mapsto o_2] \\ l_1, l_2 \in Loc \setminus \mathsf{dom}(\mu_2'),\ l_1 \neq l_2 \end{array}\right\}\right). \tag{18}
$$

Since l_2 is not used in ϕ_2 nor in μ_2', (18) becomes

$$
\begin{aligned}
&\alpha_{\tau|_{-res}}(\{\phi_2|_{-res} \star \mu_2 \in \Sigma_{\tau|_{-res}} \mid \phi_2 \in \overline{\phi}_{\tau'}(e_2),\ \mu_2 \in \overline{\mu}(e_2)\}) \\
\text{(Definition 64)} &= \alpha_{\tau|_{-res}}(\{\phi \star \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(e_2),\ \mu \in \overline{\mu}(e_2)\}) \\
\text{(Lemma 66)} &= \delta_{\tau|_{-res}}(e_2)\ .
\end{aligned}
$$

$\underline{\cup}$
By additivity (Proposition 32), the best approximation of $\cup$ over $\wp(\Sigma_\tau)$ is $\cup$ over $\wp(\Pi)$. $\square$

9.2 Proof of Propositions 48 and 51 in Section 5

To prove Proposition 48, we need some preliminary definitions and results.

Let $s \in \mathit{Frame}_\tau^{\mathcal{ER}} \times \mathit{Memory}^{\mathcal{ER}}$. We define frames and memories which use all possible creation points allowed by s.

DEFINITION 72. *Let $\phi \in \mathit{Frame}_\tau^{\mathcal{ER}}$, $\mu \in \mathit{Memory}^{\mathcal{ER}}$ and $l : \Pi \mapsto \mathit{Loc}$ be one-to-one. We define*

$$\overline{\phi}_\tau = \left\{ \phi^\flat \in \mathit{Frame}_\tau \;\middle|\; \begin{array}{l} \text{for every } v \in \mathsf{dom}(\tau) \\ \ \text{if } \tau(v) = \mathit{int} \text{ then } \phi^\flat(v) = 0 \\ \ \text{if } \tau(v) \in \mathcal{K} \text{ and } \phi(v) = \varnothing \text{ then } \phi^\flat(v) = \mathit{nil} \\ \ \text{if } \tau(v) \in \mathcal{K} \text{ and } \phi(v) \neq \varnothing \text{ then } \phi^\flat(v) \in l\phi(v) \end{array} \right\},$$

$$\overline{\mu} = \left\{ \mu^\flat \in \mathit{Memory} \;\middle|\; \begin{array}{l} \mathsf{dom}(\mu^\flat) = \mathsf{rng}(l),\ \mu^\flat(l(\pi)) = \pi \star \phi^\flat_\pi \\ \text{with } \phi^\flat_\pi \in \overline{\mu}_{F(\pi)} \text{ for every } \pi \in \Pi \end{array} \right\}.$$

Lemma 73 is needed in the proof of Lemma 74.

LEMMA 73. *Let $\phi \in \mathit{Frame}_\tau^{\mathcal{ER}}$, $\mu \in \mathit{Memory}^{\mathcal{ER}}$, $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$. Then $\varepsilon_\tau(\phi^\flat \star \mu^\flat) \subseteq \phi$.*

PROOF. For every $v \in \mathsf{dom}(\tau)$ we have

$$\begin{aligned}
\varepsilon_\tau(\phi^\flat \star \mu^\flat)(v) &= \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{(\mu^\flat\phi^\flat(v)).\pi\} & \text{if } \tau(v) \in \mathcal{K} \text{ and } \phi^\flat(v) \in \mathit{Loc} \\ \varnothing & \text{otherwise} \end{cases} \\
(\text{Definition 72}) &= \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{\mu^\flat(l(\pi')).\pi\} & \text{if } \tau(v) \in \mathcal{K} \text{ and } \phi^\flat(v) \in \mathit{Loc}, \text{ with } \pi' \in \phi(v) \\ \varnothing & \text{otherwise} \end{cases} \\
&= \begin{cases} * & \text{if } \tau(v) = \mathit{int} \\ \{\pi'\} & \text{if } \tau(v) \in \mathcal{K} \text{ and } \phi^\flat(v) \in \mathit{Loc}, \text{ with } \pi' \in \phi(v) \\ \varnothing & \text{otherwise} \end{cases} \\
&\subseteq \phi(v)\ .
\end{aligned}$$

$\square$

We prove now some properties of the frames and memories of Definition 72.

LEMMA 74. *Let $\phi \in \mathit{Frame}_\tau^{\mathcal{ER}}$, $\mu \in \mathit{Memory}^{\mathcal{ER}}$, $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$. Then*
i) $\phi^\flat \star \mu^\flat : \tau$;
ii) $\phi^\flat \star \mu^\flat \in \Sigma_\tau$ *if and only if* $\mathtt{this} \notin \mathsf{dom}(\tau)$ *or* $\phi(\mathtt{this}) \neq \varnothing$;
iii) If $\phi^\flat \star \mu^\flat \in \Sigma_\tau$ *then* $\alpha_\tau(\phi^\flat \star \mu^\flat) \subseteq \phi \star \mu$.

PROOF.

i) Condition 1 of Definition 14 holds since $\mathsf{rng}(\phi^\flat) \cap Loc \subseteq \mathsf{rng}(l) = \mathsf{dom}(\mu^\flat)$. Moreover, if $v \in \mathsf{dom}(\phi^\flat)$ and $\phi^\flat(v) \in Loc$ then $\phi^\flat(v) \in l\phi(v)$. Thus there exists $\pi \in \phi(v)$ with $(\mu^\flat\phi^\flat(v)).\pi = \pi$ and $k((\mu^\flat\phi^\flat(v)).\pi) = k(\pi) \leq \tau(v)$. Condition 2 holds since if $o \in \mathsf{rng}(\mu^\flat)$ then $o.\phi = \phi^\flat_\pi$ for some $\pi \in \Pi$. Since $\phi^\flat_\pi \in \overline{\mu}_{F(\pi)}$, reasoning as above we have that $\phi^\flat_\pi$ is weakly $F(\pi)$-correct *w.r.t.* $\mu^\flat$. Then $\phi^\flat \star \mu^\flat : \tau$.

ii) By point i, we know that $\phi^\flat \star \mu^\flat : \tau$. From Definition 16, we have $\phi^\flat \star \mu^\flat \in \Sigma_\tau$ if and only if $\mathtt{this} \notin \mathsf{dom}(\tau)$ or $\phi^\flat(\mathtt{this}) \neq nil$. By Definition 72, the latter case holds if and only if $\phi(\mathtt{this}) \neq \varnothing$.

iii) By Definition 42 we have

$$\begin{aligned} \alpha_\tau(\phi^\flat \star \mu^\flat) &= \varepsilon_\tau(\phi^\flat \star \mu^\flat) \star \varepsilon_{\overline{\tau}}(\{\overline{o.\phi} \star \mu^\flat \mid o \in O_\tau(\phi^\flat \star \mu^\flat)\}) \\ \text{(Lemma 73)} &\subseteq \phi \star \varepsilon_{\overline{\tau}}(\{\overline{o.\phi} \star \mu^\flat \mid o \in O_\tau(\phi^\flat \star \mu^\flat)\}) \ . \end{aligned}$$

By Definition 72, for every $o \in O_\tau(\phi^\flat \star \mu^\flat)$ we have $o.\phi \in \overline{\mu}_{F(\pi)}$ and hence $\overline{o.\phi} \subseteq \phi'$ with $\phi' \in \overline{\mu}_{\overline{\tau}}$. Then we have $\varepsilon_{\overline{\tau}}(\overline{o.\phi} \star \mu^\flat) \subseteq \varepsilon_{\overline{\tau}}(\phi' \star \mu^\flat)$, which by Lemma 73 is contained in μ.

□

Lemma 75 states that, given an abstract state s, if a creation point π belongs to $\rho^i(s)$ then there is a concrete state σ from those in Definition 72 and an object in $O^i(\sigma)$ created in π, and vice versa. In other words, $\rho^i(s)$ collects all and only the creation points of the objects which can ever be reached in a concrete state approximated by s.

LEMMA 75. *Let $\phi \in Frame^{\mathcal{ER}}_\tau$ be such that if $\mathtt{this} \in \mathsf{dom}(\tau)$ then $\phi(\mathtt{this}) \neq \varnothing$, $\mu \in Memory^{\mathcal{ER}}$ and $i \in \mathbb{N}$. Then $\pi \in \rho^i_\tau(\phi \star \mu)$ if and only if there exist $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$ such that $\pi = o.\pi$ for a suitable $o \in O^i_\tau(\phi^\flat \star \mu^\flat)$.*

PROOF. We proceed by induction on i. If $i = 0$ the result holds since $\rho^0_\tau(\phi \star \mu) = \varnothing$ and for every $\phi^\flat \in \overline{\phi}_\tau$ and $\mu \in \overline{\mu}$ we have $O^0_\tau(\phi^\flat \star \mu^\flat) = \varnothing$. Assume that it holds for a given $i \in \mathbb{N}$. We have $\pi \in \rho^{i+1}_\tau(\phi \star \mu)$ if and only if $\pi \in \phi(v)$ with $v \in \mathsf{dom}(\tau)$ (and hence $\tau(v) \in \mathcal{K}$) or $\pi \in \rho^i_{F(\pi)}(\phi|_{\mathsf{dom}(F(\pi'))} \star \mu)$ with $v \in \mathsf{dom}(\tau)$ and $\pi' \in \phi(v)$ (and hence $\tau(v) \in \mathcal{K}$). The first case holds if and only if $o.\pi = \pi$ with $o = \mu^\flat\phi^\flat(v)$, $v \in \mathsf{dom}(\tau)$ and $\phi^\flat(v) \in Loc$ for suitable $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$ (Definition 72). By inductive hypothesis, the second case holds if and only if there exist $\phi^\flat_1 \in \overline{\phi}_{F(\pi')}$ and $\mu^\flat \in \overline{\mu}$ such that $\pi = o.\pi$ for a suitable $o \in O^i_{F(\pi')}(\phi^\flat_1 \star \mu^\flat)$, if and only if (Definition 72) there exist $\phi^\flat \in \overline{\phi}_\tau$ and $\mu^\flat \in \overline{\mu}$ such that $\pi = o.\pi$, $v \in \mathsf{dom}(\tau)$, $\phi^\flat(v) \in Loc$, $o' = \mu^\flat\phi^\flat(v)$ and $o \in O^i_{F(o'.\pi)}(o'.\phi \star \mu^\flat)$. Together, the first or the second case hold if and only if there exist $\phi^\flat \in \overline{\phi}_\tau$ and $\mu \in \overline{\mu}$ such that $o \in O^{i+1}_\tau(\phi^\flat \star \mu^\flat)$ and $o.\pi = \pi$ (Definition 21). □

Lemma 76 says that the concrete states constructed through the frames and memories of Definition 72 represents a worst-case *w.r.t.* the set of creation points of the objects reachable in every concrete state.

LEMMA 76. *Let $\phi \star \mu \in \Sigma_\tau$, $i \in \mathbb{N}$ and $\phi^\# \star \mu^\# = \alpha^{\mathcal{ER}}_\tau(\phi \star \mu)$. If $o \in O^i_\tau(\phi \star \mu)$ then there exist $\phi^\flat \in \overline{\phi^\#}_\tau$ and $\mu^\flat \in \overline{\mu^\#}$ such that $o' \in O^i_\tau(\phi^\flat \star \mu^\flat)$ and $o'.\pi = o.\pi$.*

Proof. We proceed by induction on i. We have $O^0_\tau(\phi\star\mu)=\varnothing$ and the result holds for $i=0$. Assume that it holds for a given $i\in\mathbb{N}$. Let $o\in O^{i+1}_\tau(\phi\star\mu)$. We have $o=\mu\phi(v)$ with $v\in\mathsf{dom}(\tau)$ and $\phi(v)\in \mathit{Loc}$ or $o\in O^i_{F(o'.\pi)}(o'.\phi\star\mu)$ with $v\in\mathsf{dom}(\tau)$, $\phi(v)\in \mathit{Loc}$ and $o'=\mu\phi(v)$. In the first case, we have $o.\pi\in\phi^\#(v)$ and there exist $\phi^\flat\in\overline{\phi^\#}_\tau$ and $\mu^\flat\in\overline{\mu^\#}$ such that $\mu^\flat\phi^\flat(v).\pi=\pi$ and the thesis follows by letting $o'=\mu^\flat\phi^\flat(v)$. In the second case, by inductive hypothesis we know that there exist $\phi^\flat_1\in\overline{\phi^\#}_{F(o'.\pi)}$ and $\mu^\flat\in\overline{\mu^\#}$ such that $o''\in O^i_{F(o'.\pi)}(\phi^\flat_1\star\mu^\flat)$, $o''.\pi=o.\pi$, $v\in\mathsf{dom}(\tau)$, $\phi(v)\in \mathit{Loc}$ and $o'=\mu\phi(v)$ if and only if (Definitions 72 and 21) there exist $\phi^\flat\in\overline{\phi^\#}_\tau$ and $\mu^\flat\in\overline{\mu^\#}$ such that $o''\in O^{i+1}_\tau(\phi^\flat\star\mu^\flat)$ and $o''.\pi=o.\pi$. □

Lemma 77 gives an explicit definition of the abstraction of the set of states constructed from the frames and memories of Definition 72.

Lemma 77. *Let $\phi\in \mathit{Frame}^{\mathcal{ER}}_\tau$ and $\mu\in \mathit{Memory}^{\mathcal{ER}}$. Then*

$$\alpha^{\mathcal{ER}}_\tau(\{\phi^\flat\star\mu^\flat\in\Sigma_\tau\mid \phi^\flat\in\overline{\phi}_\tau \text{ and } \mu^\flat\in\overline{\mu}\})=\xi_\tau(\phi\star\mu)\ .$$

Proof. Let $A_\tau=\alpha^{\mathcal{ER}}_\tau(\{\phi^\flat\star\mu^\flat\in\Sigma_\tau\mid\phi^\flat\in\overline{\phi}_\tau \text{ and } \mu^\flat\in\overline{\mu}\})$. If $\mathtt{this}\in\mathsf{dom}(\tau)$ and $\phi(\mathtt{this})=\varnothing$, then $A_\tau=\bot$ because of Lemma 74.ii. Moreover, $\xi_\tau(\phi\star\mu)=\bot$ (Definition 45). Otherwise, by Definition 72 we have

$$\begin{aligned}
A_\tau &= \epsilon_\tau\left(\left\{\phi^\flat\star\mu^\flat \,\middle|\, \begin{array}{l}\phi^\flat\in\overline{\phi}_\tau\\ \mu^\flat\in\overline{\mu}\end{array}\right\}\right)\star\epsilon_{\overline{\tau}}\left(\left\{\overline{o.\phi}\star\mu^\flat \,\middle|\, \begin{array}{l}\phi^\flat\in\overline{\phi}_\tau,\ \mu^\flat\in\overline{\mu},\\ o\in O_\tau(\phi^\flat\star\mu^\flat)\end{array}\right\}\right)\\
&=\phi\star\epsilon_{\overline{\tau}}(\{\overline{\phi'}\star\mu^\flat\mid\phi'\in\overline{\mu}_{F(o.\pi)},\ \phi^\flat\in\overline{\phi}_\tau,\ \mu^\flat\in\overline{\mu},\ o\in O_\tau(\phi^\flat\star\mu^\flat)\})\\
&=\phi\star\epsilon_{\overline{\tau}}(\{\overline{\phi'}\star\mu'\mid\phi'\in\overline{\mu}_{F(o.\pi)},\ \phi^\flat\in\overline{\phi}_\tau,\ \mu',\mu^\flat\in\overline{\mu},\ o\in O_\tau(\phi^\flat\star\mu^\flat)\})
\end{aligned}\tag{19}$$

since $\epsilon_{\overline{\tau}}$ does not depend on the frames of the objects in memory (Definition 38). By Lemma 75, (19) is equal to

$$\begin{aligned}
&\phi\star\epsilon_{\overline{\tau}}(\{\overline{\phi'}\star\mu'\mid\phi'\in\overline{\mu}_{F(\pi)},\ \mu'\in\overline{\mu},\ \pi\in\rho_\tau(\phi\star\mu)\})\rangle\\
&=\phi\star\cup\{\mu|_{\mathsf{dom}(F(\pi))}\mid\pi\in\rho_\tau(\phi\star\mu)\}\cup\Im(\overline{\tau})\\
&=\xi_\tau(\phi\star\mu)\ .
\end{aligned}$$

□

We now prove Proposition 48. To do this, w will use the set of states constructed from the frames and memories in Definition 72 to show that $\alpha^{\mathcal{ER}}$ is onto.

$\boxed{\textbf{Proof} \textit{ of Proposition 48 at page 27}}$

∘ Let $X\subseteq\Sigma_\tau$. By Proposition 47, Lemmas 76 and 77 and Definition 42, we have

$$\begin{aligned}
\alpha^{\mathcal{ER}}_\tau(X)=\cup\{\alpha^{\mathcal{ER}}_\tau(\sigma)\mid\sigma\in X\}&\subseteq\cup\ \alpha^{\mathcal{ER}}_\tau\left(\left\{\phi^\flat\star\mu^\flat\in\Sigma_\tau \,\middle|\, \begin{array}{l}\phi^\flat\in\overline{\alpha^{\mathcal{ER}}_\tau(\sigma).\phi}_\tau\\ \mu^\flat\in\overline{\alpha^{\mathcal{ER}}_\tau(\sigma).\mu}\\ \sigma\in X\end{array}\right\}\right)\\
&=\cup\{\xi_\tau\alpha^{\mathcal{ER}}_\tau(\sigma)\mid\sigma\in X\}\subseteq\xi_\tau\alpha^{\mathcal{ER}}_\tau(X)\ .
\end{aligned}$$

The opposite inclusion holds since ξ_τ is reductive (Proposition 47) and, hence $\alpha^{\mathcal{ER}}_\tau(X)\in\mathsf{fp}(\xi_\tau)$. Conversely, let $s\in\mathsf{fp}(\xi_\tau)$ and $X=\{\phi^\flat\star\mu^\flat\in\Sigma_\tau\mid\phi^\flat\in$

$\overline{\phi}_\tau,\ \mu^\flat \in \overline{\mu}\}$. By Lemma 77 and since $s \in \mathsf{fp}(\xi_\tau)$, we have $\alpha_\tau^{\mathcal{ER}}(X) = \xi_\tau(s) = s$. □

The proof of Proposition 51 requires some preliminary results.

Corollary 78 states that if we know that the approximation of a set of concrete states S is some $\phi \star \mu$, then we can conclude that a better approximation of S is $\xi(\phi \star \mu)$. In other words, garbage is not used in the approximation.

Corollary 78. *Let $S \subseteq \Sigma_\tau$, $\phi \in Frame_\tau^{\mathcal{ER}}$ and $\mu \in Memory^{\mathcal{ER}}$. Then $\alpha_\tau(S) \subseteq \xi_\tau(\phi \star \mu)$ if and only if $\alpha_\tau(S) \subseteq \phi \star \mu$.*

Proof. Assume that $\alpha_\tau(S) \subseteq \xi_\tau(\phi \star \mu)$. By reductivity (Proposition 47) we have $\alpha_\tau(S) \subseteq \phi \star \mu$. Conversely, assume that $\alpha_\tau(S) \subseteq \phi \star \mu$. By Proposition 48 and monotonicity (Proposition 47) we have $\alpha_\tau(S) = \xi_\tau \alpha_\tau(S) \subseteq \xi_\tau(\phi \star \mu)$. □

The following lemma will be used in the proof of Proposition 51. It states that the approximation of a variable depends from the concrete value of that variable only, and that the approximation of a memory is the same if the locations in the frame do not change (although they may be bound to different variables).

Lemma 79. *Let $\phi' \star \mu \in \Sigma_{\tau'}$ and $\phi'' \star \mu \in \Sigma_{\tau''}$. Then*

i) if $\phi'(v) = \phi''(v)$ for each $v \in \mathsf{dom}(\tau') \cap \mathsf{dom}(\tau'')$, then $(\alpha_{\tau'}(\phi' \star \mu)).\phi(v) = (\alpha_{\tau''}(\phi'' \star \mu)).\phi(v)$;

ii) if $\mathsf{rng}(\phi') \cap Loc = \mathsf{rng}(\phi'') \cap Loc$, then $(\alpha_{\tau'}(\phi' \star \mu)).\mu = (\alpha_{\tau''}(\phi'' \star \mu)).\mu$.

Proof. From Definition 42. □

Lemma 80 says that if we consider all the concrete states approximated by some $\phi^\# sep \mu^\#$ and we restrict their frames, the resulting set of states is approximated by $\xi(\phi^\# \star \mu^\#)$. In other words, the operation ξ garbage collects all objects that, because of the restriction, are not longer reachable.

Lemma 80. *Let $vs \subseteq \mathsf{dom}(\tau)$ and $\phi^\# \star \mu^\# \in \mathcal{ER}_\tau$. Then*

$$\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) = \xi_{\tau|_{-vs}}(\phi^\#|_{-vs} \star \mu^\#) \ .$$

Proof. We have

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) \\ &= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) \ , \end{aligned} \tag{20}$$

since if $\phi \star \mu \in \Sigma_\tau$ then $\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}}$. We have that, if $\alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#$, then $\alpha_{\tau|_{-vs}}(\phi|_{-vs} \star \mu) \subseteq \phi^\#|_{-vs} \star \mu^\#$. Hence (20) is contained in $\phi^\#|_{-vs} \star \mu^\#$. By Corollary 78, (20) is also contained in $\xi_{\tau|_{-vs}}(\phi^\#|_{-vs} \star \mu^\#)$. But also the converse inclusion holds, since in (20) we can restrict the choice of $\phi \star \mu \in \Sigma_\tau$, so that (20) contains

$$\alpha_{\tau|_{-vs}}\left(\left\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \left| \begin{array}{l} \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \\ \phi \in \overline{\phi^\#}_\tau,\ \mu \in \overline{\mu^\#} \end{array} \right.\right\}\right) \ . \tag{21}$$

By points ii and iii of Lemma 65, (21) is equal to

$$
\begin{aligned}
& \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{\phi^{\#}}_{\tau},\ \mu \in \overline{\mu^{\#}}\}) \\
\text{(Definition 72)} &= \alpha_{\tau|_{-vs}}(\{\phi \star \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \in \overline{(\phi^{\#}|_{-vs})}_{\tau|_{-vs}} \text{ and } \mu \in \overline{\mu^{\#}}\}) \\
\text{(Lemma 77)} &= \xi_{\tau|_{-vs}}(\phi^{\#}|_{-vs} \star \mu^{\#}) \ .
\end{aligned}
$$

□

We are now ready to prove the correctness and optimality of the abstract operations in Figure 8.

Proof *of proposition 51 at page 29*

◦ The strictness of the abstract operations (except $\cup$) follows using similar reasoning to that of the proof of strictness in Proposition 33. Note that, $\gamma_\tau(\bot) = \varnothing$ for all $\tau \in \mathit{TypEnv}$, since, by Definition 42,

$$\gamma_\tau(\bot) = \{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \bot\} = \{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) = \bot\} = \varnothing.$$

Hence return is also strict on both arguments.

We will use the corresponding versions of the properties P2 and P3 already used in the proof of Proposition 33. They are

P2 If $\phi \star \mu \in \mathcal{E}_\tau$ then there exists $\sigma \in \Sigma_\tau$ such that $\alpha_\tau(\sigma) \subseteq \phi \star \mu$.

P3 $\alpha_\tau \gamma_\tau$ is the identity map.

To see that P2 is a consequence of P1, let π be as defined in P1; then, letting $\sigma = [\mathtt{this} \mapsto l] \star [l \mapsto \pi \star \Im(F(\pi))]$ for some $l \in \mathit{Loc}$, $\sigma \in \Sigma_\tau$. Moreover, by Definition 23, $\alpha_\tau(\sigma) = \{\pi\} \subseteq e$ so that P2 holds. By Proposition 32, α_τ is a Galois insertion and hence, P3 holds.

P2 holds since $\phi(\mathtt{this}) \neq \varnothing$ so that there exists $\pi \in \phi(\mathtt{this})$ and hence, letting $\sigma = [\mathtt{this} \mapsto l] \star [l \mapsto \pi \star \Im(F(\pi))]$ for some $l \in \mathit{Loc}$, $\sigma \in \Sigma_\tau$. Moreover, $\alpha_\tau(\sigma) = \phi^{\bot}[\mathtt{this} \mapsto \{\pi\}] \star \mu^{\bot} \subseteq \phi \star \mu$, where $\phi^{\bot}$ and $\mu^{\bot}$ are the least elements of $\mathit{Frame}_\tau^{\mathcal{ER}}$ and $\mathit{Memory}^{\mathcal{ER}}$, respectively. By Proposition 50, α_τ is a Galois insertion and hence, P3 holds.

Most cases of the proof are similar to the corresponding cases in the proof of Proposition 33, provided we use Lemma 79 instead of Lemma 70, Lemma 80 instead of Lemma 71, Definition 45 instead of Definition 26, and we modify the syntax of the abstract elements. As an example, consider

<u>$\mathsf{get\_int}$, $\mathsf{get\_nil}$, $\mathsf{get\_var}$</u>

$$
\begin{aligned}
& \alpha_{\tau[res \mapsto int]}(\mathsf{get\_int}_\tau^i(\gamma_\tau(\phi^{\#} \star \mu^{\#}))) \\
&= \alpha_{\tau[res \mapsto int]}(\{\phi'[res \mapsto i] \star \mu' \mid \phi' \star \mu' \in \gamma_\tau(\phi^{\#} \star \mu^{\#})\}) \\
(*) &= \alpha_\tau(\{\phi' \star \mu' \mid \phi' \star \mu' \in \gamma_\tau(\phi^{\#} \star \mu^{\#})\}).\phi[res \mapsto *] \star \\
&\quad \star \alpha_\tau(\{\phi' \star \mu' \mid \phi' \star \mu' \in \gamma_\tau(\phi^{\#} \star \mu^{\#})\}).\mu \\
\text{(P3)} &= \phi^{\#}[res \mapsto *] \star \mu^{\#} \ .
\end{aligned}
$$

where point $*$ follows by Lemma 79 since $res \notin \mathsf{dom}(\tau)$ and $\tau[res \mapsto int](res) = int$. The proof is similar for get_nil and get_var.

Therefore, we only show the cases which differ significantly from the corresponding case in Proposition 33.

is_nil

Let $A = \alpha_{\tau[res\mapsto int]}(\mathsf{is\_nil}_\tau(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned} A &= \alpha_{\tau[res\mapsto int]}(\mathsf{is\_nil}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \phi^\# \star \mu^\#\})) \\ &= \alpha_{\tau[res\mapsto int]}(\{\phi[res \mapsto 1] \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) . \end{aligned}$$

By Lemma 79.i we have

$$\begin{aligned} A.\phi &= \phi^\#[res \mapsto *] \\ \text{(Definition 45)} &= \xi_{\tau[res\mapsto int]}(\phi^\#[res \mapsto *] \star \mu^\#) . \end{aligned}$$

Moreover, by Lemma 79.ii we have

$$\begin{aligned} A.\mu &= \alpha_{\tau|_{-res}}(\{\phi|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}).\mu \\ \text{(Lemma 80)} &= \xi_{\tau|_{-res}}(\phi^\#|_{-res} \star \mu^\#).\mu \\ \text{(Definition 45)} &= \xi_{\tau[res\mapsto int]}(\phi^\#[res \mapsto *] \star \mu^\#).\mu . \end{aligned}$$

put_var

Let $A = \alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned} A &= \alpha_{\tau|_{-res}}(\mathsf{put\_var}_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq \phi^\# \star \mu^\#\})) \\ &= \alpha_{\tau|_{-res}}(\{\phi[v \mapsto \phi(res)]|_{-res} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}) . \end{aligned}$$

By Lemma 79.i we have

$$\begin{aligned} A.\phi &= \phi^\#[v \mapsto \phi^\#(res)]|_{-res} \\ \text{(Definition 45)} &= \xi_{\tau|_{-res}}(\phi^\#[v \mapsto \phi^\#(res)]|_{-res} \star \mu^\#).\phi . \end{aligned}$$

Moreover, since $\mathsf{rng}(\phi[v \mapsto \phi(res)]|_{-res}) = \mathsf{rng}(\phi|_{-v})$, by Lemma 79.ii we have

$$\begin{aligned} A.\mu &= \alpha_{\tau|_{-v}}(\{\phi|_{-v} \star \mu \mid \phi \star \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\}).\mu \\ \text{(Lemma 80)} &= \xi_{\tau|_{-v}}(\phi^\#|_{-v} \star \mu^\#).\mu \\ \text{(Definition 45)} &= \xi_{\tau|_{-res}}(\phi^\#[v \mapsto \phi^\#(res)]|_{-res} \star \mu^\#).\mu . \end{aligned}$$

call

Let $p = P(\nu)|_{-\mathtt{out}}$. Recall that $\mathsf{dom}(p) = \{\iota_1, \ldots, \iota_n, \mathtt{this}\}$. Let $\tau_\prime = \tau[v_1 \mapsto \iota_1, \ldots, v_n \mapsto \iota_n, res \mapsto \mathtt{this}]$ and $\phi^\#_\prime = \phi^\#[v_1 \mapsto \iota_1, \ldots, v_n \mapsto \iota_n, res \mapsto \mathtt{this}]$. We have

$$
\begin{aligned}
&\alpha_p(\mathsf{call}_\tau^{\nu,v_1,\ldots,v_n}(\gamma_\tau(\phi^{\#}\star\mu^{\#})))\\
&=\alpha_p(\mathsf{call}_\tau^{\nu,v_1,\ldots,v_n}(\{\sigma\in\Sigma_\tau \mid \alpha_\tau(\sigma)\subseteq\phi^{\#}\star\mu^{\#}\}))\\
&=\alpha_p\left(\left\{\begin{bmatrix}\iota_1\mapsto\phi(v_1),\ldots,\iota_n\mapsto\phi(v_n),\\ \mathtt{this}\mapsto\phi(res)\end{bmatrix}\star\mu \;\middle|\; \begin{array}{l}\phi\star\mu\in\Sigma_\tau \text{ and}\\ \alpha_\tau(\phi\star\mu)\subseteq\phi^{\#}\star\mu^{\#}\end{array}\right\}\right)\\
\text{(Lemma 79)}&=\alpha_p(\{\phi|_p\star\mu \mid \phi\star\mu\in\Sigma_{\tau'} \text{ and } \alpha_{\tau'}(\phi\star\mu)\subseteq\phi^{\#}_{\prime}\star\mu^{\#}\})\\
\text{(Lemma 80)}&=\xi_p(\phi^{\#}_{\prime}|_p\star\mu^{\#})\\
&=\xi_p([\iota_1\mapsto\phi^{\#}(v_1),\ldots,\iota_n\mapsto\phi^{\#}(v_n),\mathtt{this}\mapsto\phi^{\#}(res)]\star\mu^{\#})\ .
\end{aligned}
$$

<u>new</u>

Let $\kappa=k(\pi)$ and $A=\alpha_{\tau[res\mapsto\kappa]}(\mathsf{new}_\tau^\pi(\gamma_\tau(\phi^{\#}\star\mu^{\#})))$. Since $res\notin\mathsf{dom}(\tau)$ we have

$$
\begin{aligned}
A&=\alpha_{\tau[res\mapsto\kappa]}(\mathsf{new}_\tau^\pi(\{\sigma\in\Sigma_\tau \mid \alpha_\tau(\sigma)\subseteq\phi^{\#}\star\mu^{\#}\}))\\
&=\alpha_{\tau[res\mapsto\kappa]}\left(\left\{\begin{array}{c}\phi[res\mapsto l]\star\\ \star\,\mu[l\mapsto\pi\star\Im(F(\kappa))]\end{array}\;\middle|\;\begin{array}{l}\phi\star\mu\in\Sigma_\tau,\ \alpha_\tau(\phi\star\mu)\subseteq\phi^{\#}\star\mu^{\#}\\ l\in Loc\setminus\mathsf{dom}(\mu)\end{array}\right\}\right)\ .
\end{aligned}
$$

By Lemma 79.i we have

$$
\begin{aligned}
A.\phi&=\alpha_\tau(\{\phi\star\mu \mid \phi\star\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\star\mu)\subseteq\phi^{\#}\star\mu^{\#}\}).\phi[res\mapsto\{\pi\}]\\
&=\alpha_\tau\gamma_\tau(\phi^{\#}\star\mu^{\#}).\phi[res\mapsto\{\pi\}]\\
\text{(P3)}&=\phi^{\#}[res\mapsto\{\pi\}]\ .
\end{aligned}
$$

The newly created object $o=\pi\star\Im(F(\kappa))$ is such that $o.\phi(f)=\Im(F(\kappa))(f)\in\{0,nil\}$ for every $f\in\mathsf{dom}(o.\phi)$. Hence it does not contribute to the memory component $A.\mu$ and by Lemma 79.ii we have

$$
\begin{aligned}
A.\mu&=\alpha_\tau(\{\phi\star\mu \mid \phi\star\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\star\mu)\subseteq\phi^{\#}\star\mu^{\#}\}).\mu\\
&=\alpha_\tau\gamma_\tau(\phi^{\#}\star\mu^{\#}).\mu\\
\text{(P3)}&=\mu^{\#}\ .
\end{aligned}
$$

<u>return</u>

Let $\tau'=\tau[res\mapsto P(\nu)(\mathtt{out})]$, $\tau''=P(\nu)|_{\mathtt{out}}$ and $L=\mathsf{rng}(\phi_1|_{-res})\cap Loc$.

$$
\begin{aligned}
&\alpha_{\tau'}(\mathsf{return}_\tau^\nu(\gamma_\tau(\phi_1^{\#}\star\mu_1^{\#}))(\gamma_{\tau''}(\phi_2^{\#}\star\mu_2^{\#})))\\
&=\alpha_{\tau'}(\mathsf{return}_\tau^\nu(\{\sigma_1\in\Sigma_\tau \mid \alpha_\tau(\sigma_1)\subseteq\phi_1^{\#}\star\mu_1^{\#}\})(\{\sigma_2\in\Sigma_{\tau''} \mid \alpha_{\tau''}(\sigma_2)\subseteq\phi_2^{\#}\star\mu_2^{\#}\}))\\
&=\alpha_{\tau'}\left(\left\{\phi_1|_{-res}[res\mapsto\phi_2(\mathtt{out})]\star\mu_2 \;\middle|\; \underbrace{\begin{array}{c}\phi_1\star\mu_1\in\Sigma_\tau,\ \phi_2\star\mu_2\in\Sigma_{\tau''}\\ \alpha_\tau(\phi_1\star\mu_1)\subseteq\phi_1^{\#}\star\mu_1^{\#}\\ \alpha_{\tau''}(\phi_2\star\mu_2)\subseteq\phi_2^{\#}\star\mu_2^{\#},\ \mu_1\lhd_L\mu_2\end{array}}_{Cond}\right\}\right)\\
(*)&=\underbrace{\alpha_{\tau|_{-res}}(\{\phi_1|_{-res}\star\mu_2 \mid Cond\})}_{A}\cup\underbrace{\alpha_{\tau''}(\{\phi_2\star\mu_2 \mid Cond\})[\mathtt{out}\mapsto res]}_{B}
\end{aligned}
$$

where point $*$ follows by Definition 42. Since $\alpha_{\tau''}(\phi_2 \star \mu_2) \subseteq \phi_2^\# \star \mu_2^\#$, we have $B \subseteq \phi_2^\#[\mathtt{out} \mapsto \mathit{res}] \star \mu_2^\#$. But the converse inclusion holds also, since by Lemma 74.iii we have

$$B \supseteq \alpha_{\tau''}\left(\left\{\phi_2 \star \mu_2 \;\middle|\; \begin{array}{l} \phi_1 \in \overline{\phi_1^\#}_{\tau},\ \mu_1 \in \overline{\mu_1^\#} \\ \phi_2 \in \overline{\phi_2^\#}_{\tau''},\ \mu_2 \in \overline{\mu_2^\#} \end{array}\right\}\right)[\mathtt{out} \mapsto \mathit{res}]$$

which by Lemma 77 is equal to $\phi_2^\#[\mathtt{out} \mapsto \mathit{res}] \star \mu_2^\#$. Note that the condition $\mu_1 \lhd_L \mu_2$ is satisfied by Definition 72. Since $\mathsf{dom}(\tau'') = \{\mathtt{out}\}$, we conclude that $B = [\mathit{res} \mapsto \phi_2^\#(\mathtt{out})] \star \mu_2^\#$.

With regard to A, we have

$$\begin{aligned} A &\supseteq \alpha_{\tau|_{-res}}\left\{\phi_1|_{-res} \star \mu_2 \;\middle|\; \begin{array}{l} \mathit{Cond},\ \phi_1 \in \overline{\phi_1^\#}_{\tau},\ \mu_1 \in \overline{\mu_1^\#} \\ \phi_2 = \Im(\tau''),\ \mu_2 \in \overline{\mu^\top} \end{array}\right\} \\ \text{(Lemma 74)} &= \alpha_{\tau|_{-res}}\{\phi_1|_{-res} \star \mu_2 \mid \phi_1 \in \overline{\phi_1^\#}_{\tau},\ \mu_2 \in \overline{\mu^\top}\} \\ \text{(Lemma 77)} &= \xi_{\tau|_{-res}}(\phi_1^\#|_{-res} \star \mu^\top)\ . \end{aligned} \tag{22}$$

Moreover, for every $v \in \mathsf{dom}(\tau|_{-res})$ such that $\tau(v) \in \mathcal{K}$, we have

$$\begin{aligned} A.\phi(v) &= \{o.\pi \mid \phi_1|_{-res}(v) \in \mathit{Loc},\ o = \mu_2\phi_1|_{-res}(v),\ \mathit{Cond}\} \\ \text{(since } \mu_1 \lhd_L \mu_2) &= \{o.\pi \mid \phi_1|_{-res}(v) \in \mathit{Loc},\ o = \mu_1\phi_1|_{-res}(v),\ \mathit{Cond}\} \\ &\subseteq \{(\mu_1\phi_1(v)).\pi \mid \phi_1(v) \in \mathit{Loc},\ \phi_1 \star \mu_1 \in \Sigma_\tau,\ \alpha_\tau(\phi_1 \star \mu_1) \subseteq \phi_1^\# \star \mu_1^\#\} \\ &= (\alpha_\tau\gamma_\tau(\phi_1^\# \star \mu_1^\#)).\phi(v) \\ \text{(P1)} &= \phi_1^\#(v)\ . \end{aligned}$$

We conclude that $A.\phi \subseteq \phi_1^\#|_{-res}$. Moreover, we have $A.\mu \subseteq \mu^\top$. Hence $A \subseteq \phi_1^\#|_{-res} \star \mu^\top$ which, by Corollary 78, entails $A \subseteq \xi_{\tau|_{-res}}(\phi_1^\#|_{-res} \star \mu^\top)$. Together with (22), this proves that $A = \xi_{\tau|_{-res}}(\phi_1^\#|_{-res} \star \mu^\top)$.

<u>get_field</u>

Let $\tau' = \tau[\mathit{res} \mapsto (F\tau(\mathit{res}))(f)]$ and $A = \alpha_{\tau'}(\mathsf{get\_field}_\tau^f(\gamma_\tau(\phi^\# \star \mu^\#)))$. We have

$$\begin{aligned} A &= \alpha_{\tau'}(\mathsf{get\_field}_\tau^f(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\#\})) \\ &= \alpha_{\tau'}\left(\left\{\phi[\mathit{res} \mapsto (\mu\phi(\mathit{res})).\phi(f)] \star \mu \;\middle|\; \begin{array}{l} \phi \star \mu \in \Sigma_\tau,\ \phi(\mathit{res}) \neq \mathit{nil}, \\ \alpha_\tau(\phi \star \mu) \subseteq \phi^\# \star \mu^\# \end{array}\right\}\right) \end{aligned}$$

which is $\bot$ if $\phi^\#(\mathit{res}) = \varnothing$, since in such a case the condition $\phi(\mathit{res}) \neq \mathit{nil}$ cannot be satisfied. Assume then $\phi^{\#}(res) \neq \varnothing$. We have

$$A \supseteq \alpha_{\tau'}\left(\left\{\phi[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \;\middle|\; \begin{array}{l}\phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq nil,\\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#},\\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\end{array}\right\}\right)$$

$$\text{(Definition 72)} = \alpha_{\tau'}\left(\left\{\phi[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \;\middle|\; \begin{array}{l}\phi \star \mu \in \Sigma_\tau,\\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#},\\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\end{array}\right\}\right)$$

$$\text{(Lemma 74)} = \alpha_{\tau'}(\{\phi[res \mapsto (\mu\phi(res)).\phi(f)] \star \mu \mid \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\})$$

$$\text{(Definition 72)} = \alpha_{\tau'}(\{\phi \star \mu \mid \phi \in \overline{\phi^{\#}[res \mapsto \mu(f)]}_{\tau'},\ \mu \in \overline{\mu^{\#}}\})$$

$$\text{(Lemma 77)} = \phi^{\#}[res \mapsto \mu(f)] \star \mu^{\#}\ .$$

We prove that also the converse inclusion holds. Let $x = (\mu\phi(res)).\phi(f)$. If $x \in Loc$, the object $\mu(x)$ is reachable by construction from $\phi(res)$. Hence we have

$$\begin{aligned} A.\mu &\subseteq \alpha_\tau(\{\phi \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \phi(res) \neq nil,\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#}\}).\mu \\ &\subseteq \alpha_\tau(\{\phi \star \mu \mid \phi \star \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#}\}).\mu \\ &= \alpha_\tau\gamma_\tau(\phi^{\#} \star \mu^{\#}).\mu \\ \text{(P3)} &= \mu^{\#}\ . \end{aligned}$$

If $\phi(res) \neq nil$ then $o = \mu\phi(res) \in O_\tau(\phi \star \mu)$ and $\varepsilon_{\overline{\tau}}(\overline{o.\phi} \star \mu) \subseteq \mu^{\#}$ (Definition 42). Hence, if $(\mu\phi(res)).\phi(f) \neq nil$ then $((\mu\phi(res)).\phi(f)).\pi \in \mu^{\#}(f)$. By Lemma 79 we conclude that

$$A.\phi \subseteq \phi^{\#}[res \mapsto \mu^{\#}(f)]\ .$$

<u>lookup</u>

Let $A = \alpha_\tau(\mathsf{lookup}_\tau^{m,\nu}(\gamma_\tau(\phi^{\#} \star \mu^{\#})))$. We have

$$\begin{aligned} A &= \alpha_\tau(\mathsf{lookup}_\tau^{m,\nu}(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#}\})) \\ &= \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#},\ \phi(res) \neq nil,\ M((\mu\phi(res)).\pi)(m) = \nu\})\ . \end{aligned}$$

We have $A = \bot$ if there is no $\pi \in \phi^{\#}(res)$ such that $M(\pi)(m) = \nu$, because in such a case the condition $M((\mu\phi(res)).\pi)(m) = \nu$ cannot be satisfied. Otherwise we have

$$A \supseteq \alpha_\tau\left(\left\{\phi \star \mu \in \Sigma_\tau \;\middle|\; \begin{array}{l}\alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#},\ \phi(res) \neq nil\\ M((\mu\phi(res)).\pi)(m) = \nu,\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\end{array}\right\}\right)$$

$$\text{(Definition 45)} = \alpha_\tau\left(\left\{\phi \star \mu \in \Sigma_\tau \;\middle|\; \begin{array}{l}\alpha_\tau(\phi \star \mu) \subseteq \phi^{\#} \star \mu^{\#}\\ M((\mu\phi(res)).\pi)(m) = \nu,\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\end{array}\right\}\right)$$

$$\text{(Lemma 74.iii)} = \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid M((\mu\phi(res)).\pi)(m) = \nu,\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\})$$

$$\text{(Definition 72)} = \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid (\mu\phi(res)).\pi \in S,\ \phi \in \overline{\phi^{\#}}_\tau,\ \mu \in \overline{\mu^{\#}}\})$$

where $S = \{\pi \in \phi^{\#}(res) \mid M(\pi)(m) = \nu\}$. By Definition 72 we have

$$\begin{aligned} A &\supseteq \alpha_\tau(\{\phi \star \mu \in \Sigma_\tau \mid \phi \in \overline{\phi^{\#}[res \mapsto S]}_\tau,\ \mu \in \overline{\mu^{\#}}\}) \\ \text{(Lemma 77)} &= \xi_\tau(\phi^{\#}[res \mapsto S] \star \mu^{\#})\ . \end{aligned}$$

We prove that also the converse inclusion holds. Note that if $\alpha_\tau(\phi\star\mu)\subseteq\phi^\#[res\mapsto S]\star\mu^\#$ and $\phi(res)\neq nil$ then $M((\mu\phi(res)).\pi)(m)=\nu$. Hence we have

$$\begin{aligned}A &\subseteq \alpha_\tau(\{\phi\star\mu\in\Sigma_\tau \mid \alpha_\tau(\phi\star\mu)\subseteq\phi^\#[res\mapsto S]\star\mu^\#,\ \phi(res)\neq nil\})\\ \text{(Definition 42)} &= \alpha_\tau(\{\phi\star\mu\in\Sigma_\tau \mid \alpha_\tau(\phi\star\mu)\subseteq\phi^\#[res\mapsto S]\star\mu^\#\})\\ &= \alpha_\tau\gamma_\tau(\phi^\#[res\mapsto S]\star\mu^\#)\\ \text{(P3)} &= \phi^\#[res\mapsto S]\star\mu^\# \ .\end{aligned}$$

put_field

Let $\tau''=\tau|_{-res}$ and $A=\alpha_{\tau''}(\mathsf{put\_field}_{\tau,\tau'}(\gamma_\tau(\phi_1^\#\star\mu_1^\#))(\gamma_\tau(\phi_2^\#\star\mu_2^\#)))$. We have

$$\begin{aligned}A &= \alpha_{\tau''}(\mathsf{put\_field}_{\tau,\tau'}(\{\sigma_1\in\Sigma_\tau\mid\alpha_\tau(\sigma_1)\subseteq\phi_1^\#\star\mu_1^\#\})(\{\sigma_2\in\Sigma_{\tau'}\mid\alpha_{\tau'}(\sigma_2)\subseteq\phi_2^\#\star\mu_2^\#\}))\\ &= \alpha_{\tau''}\left(\left\{\begin{array}{c|l}\phi_2|_{-res}\star\mu_2[l\mapsto\mu_2(l).\pi\star & \phi_1\star\mu_1\in\Sigma_\tau,\ \phi_2\star\mu_2\in\Sigma_{\tau'}\\ \star\mu_2(l).\phi[f\mapsto\phi_2(res)]] & \alpha_\tau(\phi_1\star\mu_1)\subseteq\phi_1^\#\star\mu_1^\#\\ & \alpha_{\tau'}(\phi_2\star\mu_2)\subseteq\phi_2^\#\star\mu_2^\#\\ & (l=\phi_1(res))\neq nil,\ \mu_1\lhd_l\mu_2\end{array}\right\}\right)\end{aligned}$$

which is $\perp$ if $\phi_1^\#(res)=\varnothing$, since in such a case the condition $\phi_1(res)\neq nil$ cannot be satisfied. Assume then that $\phi_1^\#(res)\neq\varnothing$. If no creation point in $\phi_1^\#(res)$ occurs in $\phi_2^\#|_{-res}\star\mu_2^\#$ then $\mu_2(l)\notin O_{\tau''}(\phi_2|_{-res}\star\mu_2)$. Hence the update of the content of l does not contribute to $\alpha_{\tau''}$ (Definition 42) and we have

$$A=\alpha_{\tau''}\left(\left\{\phi_2|_{-res}\star\mu_2\ \middle|\ \begin{array}{l}\phi_1\star\mu_1\in\Sigma_\tau,\ \phi_2\star\mu_2\in\Sigma_{\tau'}\\ \alpha_\tau(\phi_1\star\mu_1)\subseteq\phi_1^\#\star\mu_1^\#\\ \alpha_{\tau'}(\phi_2\star\mu_2)\subseteq\phi_2^\#\star\mu_2^\#\\ (l=\phi_1(res))\neq nil,\ \mu_1\lhd_l\mu_2\end{array}\right\}\right) .$$

Let $\phi_2\star\mu_2\in\Sigma_{\tau'}$ be such that $\alpha_{\tau'}(\phi_2\star\mu_2)\subseteq\phi_2^\#\star\mu_2^\#$. By P2, we can always find $\phi_1\star\mu_1\in\Sigma_\tau$ such that $\alpha_\tau(\phi_1\star\mu_1)\subseteq\phi_1^\#\star\mu_1^\#$. By the hypothesis $\phi_1^\#(res)\neq\varnothing$ we can assume that $(l=\phi_1(res))\neq nil$. If $\mu_1\lhd_l\mu_2$ does not hold, we can assume that $l\notin\mathsf{dom}(\mu_2)$ (up to renaming). Let $o=\mu_1(l)$. We define $\mu_2'=\mu_2[l\mapsto o.\pi\star\Im(o.\pi)]$. We have $\phi_2\star\mu_2'\in\Sigma_{\tau'}$ and, since the extra location l does not contribute to $\alpha_{\tau'}$, we have $\alpha_{\tau'}(\phi_2\star\mu_2)=\alpha_{\tau'}(\phi_2\star\mu_2')$ and $\alpha_{\tau''}(\phi_2|_{-res}\star\mu_2)=\alpha_{\tau''}(\phi_2|_{-res}\star\mu_2')$. Moreover, $\mu_1\lhd_l\mu_2$ holds by construction. We conclude that the constraints on $\phi_1\star\mu_1$ and the constraint $\mu_1\lhd\mu_2$ do not contribute to A, and we have

$$\begin{aligned}A &= \alpha_{\tau''}(\{\phi_2|_{-res}\star\mu_2\mid\phi_2\star\mu_2\in\Sigma_{\tau'},\ \alpha_{\tau'}(\phi_2\star\mu_2)\subseteq\phi_2^\#\star\mu_2^\#\})\\ \text{(Corollary 78)} &= \xi_{\tau''}(\phi_2^\#|_{-res}\star\mu_2^\#) \ .\end{aligned}$$

Otherwise, since the objects reachable from $\phi_2(res)$ belong to $O_{\tau'}(\phi_2\star\mu_2)$, by Lemma 79 we have

$$\begin{aligned}A &\subseteq \alpha_{\tau''}\left(\left\{\begin{array}{c|l}\phi_2|_{-res}\star\mu_2[l\mapsto\mu_2(l).\pi\star & \phi_2\star\mu_2\in\Sigma_{\tau'},\ \alpha_{\tau'}(\phi_2\star\mu_2)\subseteq\phi_2^\#\star\mu_2^\#\\ \star\mu_2(l).\phi[f\mapsto\phi_2(res)]] & l\in\mathsf{dom}(\mu_2),\ f\in\mathsf{dom}(F(\mu_2(l).\pi))\end{array}\right\}\right)\\ &\subseteq \phi_2^\#|_{-res}\star\mu_2^\#[f\mapsto\mu_2^\#(f)\cup\phi_2^\#(res)] \ .\end{aligned}$$

By Corollary 78 we conclude that $A \subseteq \xi_{\tau''}(\phi_2^{\#}|_{-res} \star \mu_2^{\#}[f \mapsto \mu_2^{\#}(f) \cup \phi_2^{\#}(res)])$.

$\cup$
By additivity (Proposition 50), the best approximation of $\cup$ over $\wp(\Sigma_\tau)$ is (pointwise) $\cup$ over $\mathcal{ER}$. $\square$

REFERENCES

Agrawal, G. 1999. Simultaneous Demand-Driven Data-flow and Call Graph Analysis. In *Proc. of the International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society, Oxford, UK, 453–462.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers, Principles Techniques and Tools.* Addison Wesley Publishing Company.

Armstrong, T., Marriott, K., Schachte, P., and Søndergaard, H. 1998. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming 31,* 1, 3–45.

Arnold, K., Gosling, J., and Holmes, D. 2000. *The Java^{TM} Programming Language*, Third ed. Addison-Wesley.

Bacon, D. F. and Sweeney, P. F. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*. ACM SIGPLAN Notices, vol. 31(10). ACM Press, New York, 324–341.

Blanchet, B. 1998. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages (POPL'98)*. ACM Press, San Diego, CA, USA, 25–37.

Blanchet, B. 2003. Escape Analysis for Java: Theory and Practice. *ACM TOPLAS 25(6)*, 713–775.

Bogda, J. and Hölzle, U. 1999. Removing Unnecessary Synchronization in Java. In *Proc. of OOPSLA'99*. SIGPLAN Notices, vol. 34(10). Denver, Colorado, USA, 35–46.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers 35,* 8, 677–691.

Choi, J.-D., Grove, D., Hind, M., and Sarkar, V. 1999. 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)*. Toulouse, France, 21–31.

Choi, J.-D., Gupta, M., Serrano, M. J., Sreedhar, V. C., and Midkiff, S. P. 2002. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. Tech. Rep. RC22340, IBM.

Choi, J.-D., Gupta, M., Serrano, M. J., Sreedhar, V. C., and Midkiff, S. P. 2003. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM TOPLAS 25(6)*, 876–910.

Cortesi, A., Filé, G., and Winsborough, W. 1998. The Quotient of an Abstract Interpretation. *Theoretical Computer Science 202,* 1-2, 163–192.

Cousot, P. and Cousot, R. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*. 238–252.

Cousot, P. and Cousot, R. 1992. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming 13,* 2 & 3, 103–179.

Cousot, P. and Cousot, R. 2002. Modular Static Program Analysis. In *Proceedings of Compiler Construction*, R. N. Horspool, Ed. Lecture Notes in Computer Science, vol. 2304. Springer-Verlag, Grenoble, France, 159–178.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *Proc. of ECOOP'95*, W. G. Olthoff, Ed. LNCS, vol. 952. Springer-Verlag, Århus, Denmark, 77–101.

DEUTSCH, A. 1997. On the Complexity of Escape Analysis. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, Paris, France, 358–371.

DOVIER, A., PIAZZA, C., PONTELLI, E., AND ROSSI, G. 2000. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems 22,* 5, 861–931.

GAY, D. AND STEENSGAARD, B. 2000. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *Compiler Construction, 9th International Conference, (CC'00)*, D. A. Watt, Ed. Lecture Notes in Computer Science, vol. 1781. Springer-Verlag, Berlin, 82–93.

GROVE, D. AND CHAMBERS, C. 2001. A Framework for Call Graph Construction Algorithms. *ACM TOPLAS 23(6)*, 685–746.

HILL, P. M. AND SPOTO, F. 2002a. A Foundation of Escape Analysis. In *Proc. of AMAST'02*, H. Kirchner and C. Ringeissen, Eds. LNCS, vol. 2422. Springer-Verlag, St. Gilles les Bains, La Réunion island, France, 380–395.

HILL, P. M. AND SPOTO, F. 2002b. A Refinement of the Escape Property. In *Proc. of the VMCAI'02 workshop on Verification, Model-Checking and Abstract Interpretation*, A. Cortesi, Ed. Lecture Notes in Computer Science, vol. 2294. Springer-Verlag, Berlin, Venice, Italy, 154–166.

HILL, P. M. AND SPOTO, F. 2003. Logic Programs as Compact Denotations. *Computer Languages, Systems and Structures 29,* 3 (October), 45–73.

JACOBS, B. AND POLL, E. 2003. Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science 291,* 3 (January), 329–349.

JONES, N. D. AND SØNDERGAARD, H. 1987. A Semantics-based Framework for the Abstract Interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood Ltd, 123–142.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java^{TM} Virtual Machine Specification*, Second ed. Addison-Wesley.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*. ACM SIGPLAN Notices, vol. 26(11). ACM Press, 146–161.

PARK, Y. G. AND GOLDBERG, B. 1992. Escape Analysis on Lists. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*. SIGPLAN Notices, vol. 27(7). San Francisco, California, USA, 116–127.

ROUNTEV, A., MILANOVA, A., AND RYDER, B. G. 2001. Points-to Analysis for Java Using Annotated Constraints. In *Proc. of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*. ACM SIGPLAN, vol. 36(11). Tampa, Florida, USA, 43–55.

RUF, E. 2000. Effective Synchronization Removal for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. SIGPLAN Notices, vol. 35(5). Vancouver, Britith Columbia, Canada, 208–218.

RUGGIERI, C. AND MURTAGH, T. P. 1988. Lifetime Analysis of Dynamically Allocated Objects. In *15th ACM Symposium on Principles of Programming Languages (POPL'88)*. San Diego, California, USA, 285–293.

SALCIANU, A. 2001. Pointer Analysis and Its Application to Java Programs. Ph.D. thesis, MIT.

SALCIANU, A. AND RINARD, M. 2001. Pointer and Escape Analysis for Multithreaded Programs. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*. SIGPLAN Notices, vol. 36(7). Snowbird, Utah, USA, 12–23.

SPOTO, F. 2001. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In *Proc. of the Static Analysis Symposium, SAS'01*, P. Cousot, Ed. Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Berlin, Paris, 127–145.

SPOTO, F. 2002. The LOOP Analyser. `www.sci.univr.it/~spoto/loop`.

SPOTO, F. 2004. The JULIA Generic Static Analyser. `www.sci.univr.it/~spoto/julia`.

SPOTO, F. AND JENSEN, T. 2003. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS) 25,* 5 (September), 578–630.

Streckenbach, M. and Snelting, G. 2000. Points-to for Java: A General Framework and an Empirical Comparison. Tech. rep., Universit"at Passau, Germany. November.

Tip, F. and Palsberg, J. 2000. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proc. of OOPSLA'00*. SIGPLAN Notices, vol. 35(10). ACM, Minneapolis, Minnesota, USA, 281–293.

Vivien, F. and Rinard, M. 2001. Incrementalized Pointer and Escape Analysis. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. SIGPLAN Notices, vol. 36(5). Snowbird, Utah, USA, 35–46.

Whaley, J. and Lam, M. S. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proc. of ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*, W. Pugh and C. Chambers, Eds. ACM, Washington, DC, USA, 131–144.

Whaley, J. and Rinard, M. C. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*. SIGPLAN Notices, vol. 34(1). Denver, Colorado, USA, 187–206.