A Refinement of the Escape Property*

Patricia M. Hill[1] and Fausto Spoto[2]

[1] School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk
[2] Dipartimento di Informatica, Verona, Italy
spoto@sci.univr.it
Ph.: +44 01132336807   Fax: +44 01132335468

Abstract Escape analysis of object-oriented languages determines, for every program point, the *escape property* $\mathcal{E}$ *i.e.,* the set of the creation points of the objects reachable from some variables. An approximation of $\mathcal{E}$ is useful to stack allocate dynamically created objects and to reduce the overhead of synchronisation in Java-like languages. $\mathcal{E}$ can itself be used for escape analysis, but it is very imprecise. We define here a refinement $\mathcal{ER}$ of $\mathcal{E}$, in the sense that $\mathcal{ER}$ is more concrete than $\mathcal{E}$ and, hence, leads to a more precise escape analysis than $\mathcal{E}$.

1 Introduction

Escape analysis should determine which dynamically created data structures will not *escape* from their creating method. This information allows us to allocate these structures to the stack instead of the heap. Compared to heap allocation, stack allocation reduces the garbage collection overhead at run-time. Moreover, in the case of object-oriented languages such as Java, a knowledge of those objects that cannot *escape* the methods of their creating threads can be used to remove unnecessary synchronisations when the objects are accessed.

In [8], an escape analysis for object-oriented languages is defined on an *escape property* $\mathcal{E}$ which is an abstract interpretation of concrete states. This property collects, for each program point, the set of creation points of objects reachable at that point from the variables and fields in scope. That analysis, although very imprecise, was shown in [8] to be sometimes more precise than other escape analyses proposed in literature [3,4,7,12,15]. Hence, in [8] we concluded that these other analyses are not a concretisation of $\mathcal{E}$ itself and that, for improved precision, a *refinement* of $\mathcal{E}$ *i.e.,* a more concrete domain than $\mathcal{E}$, should be constructed. This paper defines the refinement $\mathcal{ER}$ by splitting the sets of creation points in $\mathcal{E}$ into subsets, one for each variable or field. Our implementation of $\mathcal{ER}$ confirms that it is indeed more precise than $\mathcal{E}$.

2 Overview

We illustrate our domain $\mathcal{ER}$ through its implementation inside a static analyser for simple object-oriented languages, called LOOP [13]. Escape analysis is typi-

* This work has been funded by EPSRC grant GR/R53401.

```
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;
method draw() : void is
  % put something here...
```

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

class main :
method main(n : figure) : void is
  f : figure;
  f := new square;    {π2}
  f.def();
  rotate(f);          {w1}
  f := new circle;    {π3}
  f.def();
  f.next = n;
  while(f)
    rotate(f);        {w2}
    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();
```

Figure 1. An example of program.

cally used to stack allocate objects which do not *escape* their creating method. Thus we direct the analysis of LOOP over the program points immediately after a call to a method which creates the data structure we want to stack allocate.

To see how the domain $\mathcal{ER}$ can be used for escape analyses, consider the program in Figure 1. We assume that `main` is called with `n` bound to a list of `circles` and we direct the analysis on the program points w_1 and w_2 which follow a `rotate(f)` call. If the creation point π_4 is not reachable from `f` in w_1 or w_2, then the `angles` created by that call can be stack allocated. With the domain $\mathcal{E}$ (Section 5), we analyse the program starting from an abstract state $\{\overline{\pi}, \pi_3\}$, the collection of all the creation points for `circles` (π_3) and for `main` ($\overline{\pi}$). With the new domain $\mathcal{ER}$ (Section 6), we analyse it instead starting from an abstract state which binds `n` and the field `next` to $\{\pi_3\}$, hence distributing the creation points over variables and fields. Using $\mathcal{E}$, LOOP computes at w_1 and w_2 the abstract information $\{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$. All we can conclude is that the creation point π_4 might be reachable in both w_1 and w_2. Using $\mathcal{ER}$, it computes:

$$w_1 : \left\langle \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \{\pi_3\}, \\ \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix}, \begin{bmatrix} \mathtt{rotation} \mapsto \{\pi_1, \pi_4\}, \\ \mathtt{next} \mapsto \{\pi_3\}, \mathtt{degree} \mapsto *, \ldots \end{bmatrix} \right\rangle \quad \text{and}$$

$$w_2 : \langle \begin{bmatrix} \mathtt{f} \mapsto \{\pi_3\}, \mathtt{n} \mapsto \{\pi_3\}, \\ \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix}, \begin{bmatrix} \mathtt{rotation} \mapsto \varnothing, \\ \mathtt{next} \mapsto \{\pi_3\}, \mathtt{degree} \mapsto *, \ldots \end{bmatrix} \rangle \ .$$

The value $*$ means that the corresponding variable is of type *int*, hence no object is bound to it. As before, we can conclude that in w_1 an object created in π_4 might be reachable from `f`. Namely, `f` is allowed to be bound to an object created in π_2 which has class `square`. This class has a field `rotation` which may contain an object created in π_4. On the other hand, in w_2 we can conclude that no object created in π_4 can be reachable from `f`. Namely, `f` can only be bound to an object created in π_3, which is hence a `circle`, and thus has no field which can be created in π_4. Note that $\mathcal{ER}$ provides even more information. Namely, it says that in w_2 *no* object created in π_4 is reachable. This does not mean that they have not been created. For instance, in w_2 an object has been created in π_4 by the call `rotate(f)` just before w_1. But that object is not reachable anymore in w_2. This information is useful for the static prediction for garbage collection.

3 Preliminaries

The (*co-*)*domain* of a function f is $\mathsf{dom}(f)$ ($\mathsf{rng}(f)$). A total (partial) function is denoted by $\mapsto$ ($\rightarrow$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ a function f whose domain is $\{v_1, \ldots, v_n\}$ and such that $f(v_i) = t_i$ for $i = 1, \ldots, n$. An *update* of f is denoted by $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain of f 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 $\mathsf{dom}(f) \subseteq \mathsf{rng}(f)$ and $f(x) = x$ then x is a *fixpoint* of f. The set of fixpoints of f is $\mathsf{fp}(f)$. A definition like $S = \langle a, b \rangle$, with a and b meta-variables, silently defines the selectors $s.a$ and $s.b$ for $s \in S$. An element x will often stand for the singleton set $\{x\}$.

A pair $\langle C, \leq \rangle$ is a *poset* if $\leq$ is reflexive, transitive and antisymmetric on C. A poset is a *complete lattice* when *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist. If $\langle C, \leq \rangle$ and $\langle A, \preceq \rangle$ are posets, then $f : C \mapsto A$ is *(co-)additive* if it preserves lub's (glb's). It is a *lower closure operator* (*lco*) if it is *monotonic*, *reductive* ($f(c) \leq c$ for every $c \in C$) and *idempotent* ($ff(c) = f(c)$ for every $c \in C$).

In abstract interpretation (AI) [5], a *Galois connection* between two posets $\langle C, \leq \rangle$ and $\langle A, \preceq \rangle$ (the *concrete* and the *abstract* domain) 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* if $\alpha\gamma$ is the identity map *i.e.,* if A does not contain *useless* elements. This is equivalent to α being onto, or γ one-to-one. The *abstraction* α and the *concretisation* γ determine each other. If C and A are complete lattices and α is additive, it is the abstraction map of a Galois connection.

4 The Framework of Analysis

In this paper we build on the *watchpoint semantics* for the static analysis of object-oriented programs [14]. That framework allows us to derive a compositional and *focused* analyser from the specification of a domain of abstract states

$$\mathcal{K} = \{\texttt{angle}, \texttt{figure}, \texttt{square}, \texttt{circle}, \texttt{main}\}$$

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

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

$$F(\texttt{square}) = \begin{bmatrix} \texttt{next} \mapsto \texttt{figure}, \texttt{side} \mapsto \mathit{int}, \texttt{squarex} \mapsto \mathit{int} \\ \texttt{squarey} \mapsto \mathit{int}, \texttt{rotation} \mapsto \texttt{angle} \end{bmatrix}$$

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

Figure 2. The static information of the program in Figure 1.

and of some operations which work over them. Then we do not have to consider any problem like scoping, recursion and name clash, since they are already solved by the watchpoint semantics. Moreover, using that framework we can measure the precision of the analysis by measuring that of the domain.

We introduce the states [9] which are the concrete domain of a version of the watchpoint semantics [14] for object-oriented programs. We refer to [9] for the concrete operations over those states and to [14] for the construction of the semantics from those states and operations. Hence some definitions contained in [9,14] will be omitted here.

We assume that we analyse programs of a simple object-oriented language, whose only basic type is *int*. All other types are called *classes*. A *typing* assigns types to a finite set of variables. The variable `this` must be bound to a class.

Definition 1. *Let Id be a set of* identifiers*, $\mathcal{K}$ a finite set of* classes *ordered by a* subclass relation $\leq$ *such that $\langle \mathcal{K}, \leq \rangle$ is a poset and* $\texttt{main} \in \mathcal{K}$*. Let Type be the set $\{\mathit{int}\} + \mathcal{K}$. We extend $\leq$ to Type by defining $\mathit{int} \leq \mathit{int}$. Let $\mathit{Vars} \subseteq \mathit{Id}$ be a set of* variables *such that* $\texttt{this} \in \mathit{Vars}$*. We define*

$$\mathit{Typing} = \{\tau : \mathit{Vars} \to \mathit{Type} \mid \mathsf{dom}(\tau) \textit{ is finite, if } \texttt{this} \in \mathsf{dom}(\tau) \textit{ then } \tau(\texttt{this}) \in \mathcal{K}\}.$$

Types and typings are initialised as $\mathsf{init}(\mathit{int}) = 0$, $\mathsf{init}(\kappa) = \mathit{nil}$ *for* $\kappa \in \mathcal{K}$ *and* $\mathsf{init}(\tau)(v) = \mathsf{init}(\tau(v))$ *for* $\tau \in \mathit{Typing}$ *and* $v \in \mathsf{dom}(\tau)$*.*

A class contains local variables (*fields*). Then *Fields* is a set of maps which bind each class to the typing of its fields. We require that different fields have different names, and that the variable `this` cannot be a field.

Definition 2. $\mathit{Fields} = \{F : \mathcal{K} \mapsto \mathit{Typing} \mid \texttt{this} \notin \mathsf{dom}(F(\kappa)) \textit{ for all } \kappa \in \mathcal{K}\}$*. We require that if* $F \in \mathit{Fields}$*,* $\kappa_1, \kappa_2 \in \mathcal{K}$ *and* $f \in \mathsf{dom}(F(\kappa_1)) \cap \mathsf{dom}(F(\kappa_2))$ *then* $F(\kappa_1)(f) = F(\kappa_2)(f)$*.*

We use the *static information* of the program to be analysed.

Definition 3. *The* static information *of a program consists of a poset* $\langle \mathcal{K}, \leq \rangle$ *and a map* $F \in \mathit{Fields}$*.*

Figures 1 and 2 show a program and its static information. The fields `x` and `y` of the classes `square` and `circle` have been disambiguated.

Definition 4. *We assume that we have a finite set Π of* creation points*. A map $k : \Pi \mapsto \mathcal{K}$ relates every creation point with the class of the objects it creates. A hidden creation point $\overline{\pi} \in \Pi$,* internal *to the operating system, creates objects of class* `main`*. Then we define $k(\overline{\pi}) =$* `main`*. Every other $\pi \in \Pi$ decorates a* `new` *κ statement in the program. Then we define $k(\pi) = \kappa$. Let $\pi \in \Pi$ and $F \in$ Fields. We define $F(\pi) = F(k(\pi))$.*

Look at Figure 3. *Locations* are references to memory cells. *Values* are integers, locations or *nil*. *Frame* is a set of maps from variables to values. The values must be consistent with the type of the variables. For instance, class variables must be bound to a location or to *nil*. An *object* contains its creation point and the frame of its fields. *Memory* is a set of maps from locations to objects. As in [8], we use here a more concrete notion of object than in [9], since they have a creation point, essential for the subsequent abstraction into a domain for escape analysis.

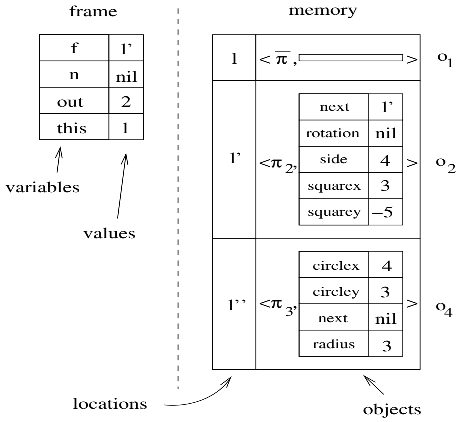

Figure 3. Frame ϕ_1 and memory μ_1 for $\tau = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{n} \mapsto \mathtt{figure}, \mathtt{out} \mapsto int, \mathtt{this} \mapsto \mathtt{main}]$.

Definition 5. *Let Loc be an infinite set of* locations*, Value $= \mathbb{Z} + Loc + \{nil\}$ and $\tau \in$ Typing. We define* frames*,* objects *and* memories *as*

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

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

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

Example 1. Let τ be as in Figure 3 *i.e.,* the typing at program point w_1 in Figure 1. Let $l, l' \in Loc$. The set $Frame_\tau$ contains $\phi_1 = [\mathtt{f} \mapsto l', \mathtt{n} \mapsto nil, \mathtt{out} \mapsto 2, \mathtt{this} \mapsto l]$ (Figure 3), but it does not contain $\phi_2 = [\mathtt{f} \mapsto 2, \mathtt{n} \mapsto l', \mathtt{out} \mapsto -2, \mathtt{this} \mapsto l]$, because `f` is bound to 2 in ϕ_2 (while it has class `figure` in τ).

Example 2. An object o_1 created at the hidden creation point $\overline{\pi}$ has class $k(\overline{\pi}) =$ `main` (Definition 4). Since $F(\mathtt{main}) = []$ (Figure 2), we have $o_1 = \langle \overline{\pi}, [] \rangle$ (Figure 3). Objects created at π_2 have class $k(\pi_2) =$ `square` (Figure 1). Examples of such objects (where $l, l' \in Loc$), consistent with $F(\mathtt{square})$, are (Figure 3)

$$o_2 = \langle \pi_2, [\mathtt{next} \mapsto l', \mathtt{rotation} \mapsto nil, \mathtt{side} \mapsto 4, \mathtt{squarex} \mapsto 3, \mathtt{squarey} \mapsto -5] \rangle,$$
$$o_3 = \langle \pi_2, [\mathtt{next} \mapsto nil, \mathtt{rotation} \mapsto l, \mathtt{side} \mapsto 4, \mathtt{squarex} \mapsto 3, \mathtt{squarey} \mapsto -5] \rangle.$$

Objects created at π_3 have class $k(\pi_3) =$ `circle` (Figure 1). An example of such objects, consistent with $F(\mathtt{circle})$, is (Figure 3)

$$o_4 = \langle \pi_3, [\mathtt{circlex} \mapsto 4, \mathtt{circley} \mapsto 3, \mathtt{next} \mapsto nil, \mathtt{radius} \mapsto 3] \rangle\ .$$

Example 3. Let $l, l', l'' \in \mathit{Loc}$ be distinct and o_1, o_2, o_3, o_4 from Example 2. Then *Memory* contains (for μ_1, see Figure 3)

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

We define a notion of type correctness which guarantees that variables are bound to locations which contain objects allowed by the type of the variables.

Definition 6. *Let $\tau \in \mathit{Typing}$, $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is* weakly τ-correct *w.r.t. μ if for every $v \in \mathsf{dom}(\phi)$ such that $\phi(v) \in \mathit{Loc}$ we have $\phi(v) \in \mathsf{dom}(\mu)$ and $k((\mu\phi(v)).\pi) \leq \tau(v)$.*

We strengthen the correctness notion of Definition 6 by requiring that it holds for the fields of the objects in memory also.

Definition 7. *Let $\tau \in \mathit{Typing}$, $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is* τ-correct *w.r.t. μ, and we write $(\phi, \mu) : \tau$, if*

1. *ϕ is weakly τ-correct w.r.t. μ (Definition 6),*
2. *for every $o \in \mathsf{rng}(\mu)$ we have that $o.\phi$ is weakly $F(o.\pi)$-correct w.r.t. μ.*

Example 4. Let τ, ϕ_1 and μ_1 be as in Figure 3 and μ_2 and μ_3 as in Example 3.

- $(\phi_1, \mu_1) : \tau$ (Figure 3). Condition 1 of Definition 7 holds because $\{v \in \mathsf{dom}(\phi_1) \mid \phi_1(v) \in \mathit{Loc}\} = \{\mathtt{this}, \mathtt{f}\}$, $\{l, l'\} \subseteq \mathsf{dom}(\mu_1)$, $k(\mu_1(l).\pi) = k(o_1.\pi) = k(\overline{\pi}) = \mathtt{main} = \tau(\mathtt{this})$ and $k(\mu_1(l').\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{square} \leq \mathtt{figure} = \tau(\mathtt{f})$. Condition 2 holds because $\mathsf{rng}(\mu_1) = \{o_1, o_2, o_4\}$, $o_1.\phi = []$, $\mathsf{rng}(o_4.\phi) \cap \mathit{Loc} = \varnothing$, $\mathsf{rng}(o_2.\phi) \cap \mathit{Loc} = \{l'\}$, $l' \in \mathsf{dom}(\mu_1)$ and $k(\mu_1(l').\pi) = \mathtt{square} \leq \mathtt{figure} = F(o_2.\pi)(\mathtt{next})$.
- $(\phi_1, \mu_2) : \tau$ does not hold, since condition 1 of Definition 7 does not hold. Namely, $\tau(\mathtt{this}) = \mathtt{main}$, $k((\mu_2\phi_1(\mathtt{this})).\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{square}$ and $\mathtt{square} \not\leq \mathtt{main}$.
- $(\phi_1, \mu_3) : \tau$ does not hold, since condition 2 of Definition 7 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 $o_3.\phi(\mathtt{rotation}) = l$, $\mathtt{square} \not\leq \mathtt{angle}$ but $k(\mu_3(l).\pi) = k(o_2.\pi) = k(\pi_2) = \mathtt{square}$ and $F(o_3.\pi)(\mathtt{rotation}) = F(\mathtt{square})(\mathtt{rotation}) = \mathtt{angle}$.

The state of the computation is a pair consisting of a frame and a memory. The variable `this` in the domain of the frame must be bound to an object.

Definition 8. *Let $\tau \in \mathit{Typing}$. We define the* states

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

Example 5. In Example 4, we have $\langle \phi_1, \mu_1 \rangle \in \Sigma_\tau$ (Figure 3) but $\langle \phi_1, \mu_2 \rangle \notin \Sigma_\tau$.

Every AI of $\wp(\Sigma_\tau)$ induces an AI of the semantics [14]. This has been used to define in [8] the escape property $\mathcal{E}$, which we briefly introduce in Section 5.

5 The Escape Property $\mathcal{E}$

Since escape analysis *over*approximates, for every program point p, the set of creation points of the objects reachable in p from some variable or field in scope, we first define a notion of *reachable* objects.

Definition 9. *Let $\tau \in$ Typing, $\sigma = \langle \phi, \mu \rangle \in \Sigma_\tau$ and $S \subseteq \Sigma_\tau$. The set of the objects* reachable *in σ is $O_\tau(\sigma) = \{O^i_\tau(\sigma) \mid i \geq 0\}$ where*

$$\begin{aligned} O^0_\tau(\sigma) &= \varnothing \\ O^{i+1}_\tau(\sigma) &= \{\mu\phi(v) \mid v \in \mathsf{dom}(\tau),\ \phi(v) \in Loc\} \cup \\ &\qquad \cup \{\mu(o.\phi(f)) \mid o \in O^i_\tau(\sigma),\ f \in \mathsf{dom}(F(o.\pi)),\ o.\phi(f) \in Loc\}\ . \end{aligned}$$

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

Note that variables and fields of type *int* do not contribute to $\alpha^{\mathcal{E}}_\tau$.

Example 6. In Figure 3, we have $O_\tau(\langle \phi_1, \mu_1 \rangle) = \{o_1, o_2\}$, $\alpha^{\mathcal{E}}_\tau(\langle \phi_1, \mu_1 \rangle) = \{\overline{\pi}, \pi_2\}$ and $o_4 \notin O_\tau(\langle \phi_1, \mu_1 \rangle)$, since o_4 is not reachable from the variables of ϕ_1.

In general, it is $\mathsf{rng}(\alpha^{\mathcal{E}}_\tau) \neq \wp(\Pi)$, since $\alpha^{\mathcal{E}}_\tau$ is not necessarily onto. This means that if we chose $\wp(\Pi)$ as abstract domain, it would contain useless elements (Section 3). We look hence only for those elements of $\wp(\Pi)$ which belongs to $\mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$. Namely, for every $S \in \wp(\Pi)$ we define $\delta_\tau(S)$ as the largest subset of S which contains only those creation points deemed useful by the typing τ. Note that if there are no possible creation points for `this`, all creation points are useless.

Definition 10. *Let $\tau \in$ Typing and $S \subseteq \Pi$. We define $\delta_\tau(S) = \cup\{\delta^i_\tau(S) \mid i \geq 0\}$ with*

$$\begin{aligned} \delta^0_\tau(S) &= \varnothing \\ \delta^{i+1}_\tau(S) &= \begin{cases} \varnothing & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and there is no } \pi \in S \text{ s.t. } k(\pi) \leq \tau(\mathtt{this}) \\ \cup\{\{\pi\} \cup \delta^i_{F(\pi)}(S) \mid \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in S,\ k(\pi) \leq \kappa\} & \text{otherwise.} \end{cases} \end{aligned}$$

Lemma 1. *Let $\tau \in$ Typing and $i \in \mathbb{N}$. The maps δ^i_τ and δ_τ are lco's.*

The following result proves that δ_τ can be used to define $\mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$.

Lemma 2. *Let $\tau \in$ Typing. Then $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha^{\mathcal{E}}_\tau)$.*

Lemma 2 allows us to assume that $\alpha^{\mathcal{E}}_\tau : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\delta_\tau)$. Moreover, it justifies the following definition of the simplest domain $\mathcal{E}$ for escape analysis. It coincides with the *escape property* itself.

Definition 11. *Let $\tau \in$ Typing. The* escape property *is $\mathcal{E}_\tau = \mathsf{fp}(\delta_\tau)$, ordered by set inclusion.*

Example 7. Let τ be as in Figure 3. We have

$$\mathcal{E}_\tau = \varnothing \cup \{S \in \wp(\Pi) \mid \overline{\pi} \in S \text{ and if } \pi_1 \in S \text{ or } \pi_4 \in S \text{ then } \pi_2 \in S\} \ .$$

The constraint on $\mathcal{E}_\tau$ says that to reach an `angle` (created in π_1 or in π_4) from the variables in $\mathsf{dom}(\tau)$, we must be able to reach a `square` (created in π_2).

Proposition 1. *Let $\tau \in$ Typing. The map $\alpha^{\mathcal{E}}_\tau$ is strict, additive and onto i.e., it is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{E}_\tau$.*

For the best approximations over $\mathcal{E}$ of the operations in [9], see [8].

6 The Refined Domain $\mathcal{ER}$

We define here a *refinement* $\mathcal{ER}$ of the domain $\mathcal{E}$ of Section 5, in the sense that $\mathcal{ER}$ is a concretisation of $\mathcal{E}$ (Proposition 3).

An abstract value (compare with Definition 5) is $*$, which approximates the integers, or $S \subseteq \Pi$, which approximates *nil* and all locations containing an object created in some creation point in S. An abstract frame maps variables to abstract values consistent with their type.

Definition 12. *Let $Value^{\mathcal{ER}} = \{*\} \cup \wp(\Pi)$ and $\tau \in$ Typing. Then*

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

Example 8. Let τ be as in Figure 3. We have

$$[\mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \{\pi_2, \pi_3\}, \mathtt{out} \mapsto \varnothing, \mathtt{this} \mapsto \{\overline{\pi}\}] \in Frame^{\mathcal{ER}}_\tau$$
$$[\mathtt{f} \mapsto \{\overline{\pi}, \pi_2\}, \mathtt{n} \mapsto \{\pi_2, \pi_3\}, \mathtt{out} \mapsto \varnothing, \mathtt{this} \mapsto \{\overline{\pi}\}] \notin Frame^{\mathcal{ER}}_\tau \ ,$$

since $k(\overline{\pi}) = \mathtt{main}$, $\tau(\mathtt{f}) = \mathtt{figure}$ and $\mathtt{main} \not\le \mathtt{figure}$.

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

Definition 13. *Let $\tau \in$ Typing. The map $\varepsilon_\tau : \wp(\Sigma_\tau) \mapsto Frame^{\mathcal{ER}}_\tau$ is such that, for every $S \subseteq \Sigma_\tau$ and $v \in \mathsf{dom}(\tau)$,*

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

Example 9. In Figure 3, we have $\varepsilon_\tau(\langle \phi_1, \mu_1 \rangle) = [\mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing, \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\}]$.

Because of Definition 2, the typing $\overline{\tau}$ *of all the fields* is well-defined.

Definition 14. *We define* $\overline{\tau} = \cup\{F(\kappa) \mid \kappa \in \mathcal{K}\}$. *Let* $\tau \in$ *Typing such that* $\mathsf{dom}(\tau) \subseteq \mathsf{dom}(\overline{\tau})$ *and* $\phi \in Frame_\tau$. *Its* extension $\overline{\phi} \in Frame_{\overline{\tau}}$ *is such that, for every* $v \in \mathsf{dom}(\overline{\tau})$,

$$\overline{\phi}(v) = \begin{cases} \phi(v) & \textit{if } v \in \mathsf{dom}(\tau) \\ \mathsf{init}(\overline{\tau}(v)) & \textit{otherwise (Definition 1).} \end{cases}$$

Example 10. Consider Figure 2. We have

$$\overline{\tau} = \begin{bmatrix} \mathtt{circlex} \mapsto \mathit{int}, \mathtt{circley} \mapsto \mathit{int}, \mathtt{degree} \mapsto \mathit{int} \\ \mathtt{next} \mapsto \mathtt{figure}, \mathtt{radius} \mapsto \mathit{int}, \mathtt{rotation} \mapsto \mathtt{angle} \\ \mathtt{side} \mapsto \mathit{int}, \mathtt{squarex} \mapsto \mathit{int}, \mathtt{squarey} \mapsto \mathit{int} \end{bmatrix} .$$

Let $\phi = [\mathtt{circlex} \mapsto 12, \mathtt{circley} \mapsto 5, \mathtt{next} \mapsto l, \mathtt{radius} \mapsto 5] \in F(\mathtt{circle})$, with $l \in Loc$. We have

$$\overline{\phi} = \begin{bmatrix} \mathtt{circlex} \mapsto 12, \mathtt{circley} \mapsto 5, \mathtt{degree} \mapsto 0, \mathtt{next} \mapsto l, \mathtt{radius} \mapsto 5 \\ \mathtt{rotation} \mapsto \mathit{nil}, \mathtt{side} \mapsto 0, \mathtt{squarex} \mapsto 0, \mathtt{squarey} \mapsto 0 \end{bmatrix} .$$

An abstract memory is an abstract frame for the collection of the fields of all classes. The abstraction map computes the abstract memory by extracting the creation points of the fields of the reachable objects of the concrete memory.

Definition 15. *Let* $\tau \in$ *Typing and* $Memory^{\mathcal{ER}} = Frame^{\mathcal{ER}}_{\overline{\tau}}$. *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* $S \subseteq \Sigma_\tau$,

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

Example 11. In Figure 3, we have (compare with Example 6. The fields not represented are implicitly bound to $*$)

$$\alpha^{\mathcal{ER}}_\tau(\langle \phi_1, \mu_1 \rangle) = \left\langle \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2\}, \mathtt{n} \mapsto \varnothing \\ \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix}, \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2\} \\ \mathtt{rotation} \mapsto \varnothing, \ldots \end{bmatrix} \right\rangle .$$

Like $\alpha^{\mathcal{E}}_\tau$ (Section 5), also the map $\alpha^{\mathcal{ER}}_\tau$ is not necessarily onto.

Example 12. Let $\tau = [\mathtt{c} \mapsto \mathtt{circle}]$. From a $\mathtt{circle}$ is not possible to reach an object of type $\mathtt{angle}$. Then there is no $\sigma \in \Sigma_\tau$ such that $\alpha^{\mathcal{ER}}_\tau(\sigma)$ is equal to the abstract state $s = \langle [\mathtt{c} \mapsto \{\pi_3\}], [\mathtt{next} \mapsto \varnothing, \mathtt{rotation} \mapsto \{\pi_1\}, \ldots] \rangle$.

Hence, we define a map ρ which collects the *reachable* creation points from an abstract frame and memory, and a map ξ which forces to $\varnothing$ the fields of type class of the objects which have no reachable creation point.

Definition 16. *Let* $\tau \in$ *Typing. We define* $\rho_\tau : Frame^{\mathcal{ER}}_\tau \times Memory^{\mathcal{ER}} \mapsto \wp(\Pi)$ *and* $\xi_\tau : \{\bot\} \cup (Frame^{\mathcal{ER}}_\tau \times Memory^{\mathcal{ER}}) \mapsto \{\bot\} \cup (Frame^{\mathcal{ER}}_\tau \times Memory^{\mathcal{ER}})$ *as* $\rho_\tau(s) = \cup\{\rho^i_\tau(s) \mid i \geq 0\}$, *where*

$$\begin{aligned} \rho^0_\tau(\langle \phi, \mu \rangle) &= \varnothing \\ \rho^{i+1}_\tau(\langle \phi, \mu \rangle) &= \{\pi \in \phi(v) \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K}\} \cup \\ &\quad \cup \{\pi \in \mu(f) \mid \pi' \in \rho^i_\tau(\langle \phi, \mu \rangle),\ f \in \mathsf{dom}(F(\pi')),\ F(\pi')(f) \in \mathcal{K}\} \end{aligned}$$

and

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

$$\xi_\tau(\langle\phi,\mu\rangle) = \begin{cases} \bot & \textit{if } \texttt{this} \in \mathsf{dom}(\tau) \textit{ and } \phi(\texttt{this}) = \varnothing \\ \langle \phi, \cup\{\mu|_{\mathsf{dom}(F(\pi))} \mid \pi \in \rho_\tau(\langle\phi,\mu\rangle)\} \cup \mathsf{init}(\overline{\tau})\rangle & \textit{otherwise.} \end{cases}$$

Example 13. In Example 12, we have $\rho_\tau(s) = \{\pi_3\}$ and hence

$$\xi_\tau(s) = \langle[\mathsf{c} \mapsto \{\pi_3\}], [\texttt{next} \mapsto \varnothing, \texttt{rotation} \mapsto \varnothing, \ldots]\rangle \ .$$

Lemma 3. *Let $\tau \in Typing$. The map ξ_τ is an lco.*

The map ξ_τ can be used to define $\mathsf{rng}(\alpha_\tau^{\mathcal{ER}})$.

Lemma 4. *Let $\tau \in Typing$. Then $\mathsf{fp}(\xi_\tau) = \mathsf{rng}(\alpha_\tau^{\mathcal{ER}})$.*

Lemma 4 allows us to assume that $\alpha_\tau^{\mathcal{ER}} : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\xi_\tau)$, and justifies the following definition.

Definition 17. *Let $\tau \in Typing$. We define $\mathcal{ER}_\tau = \mathsf{fp}(\xi_\tau)$, ordered by pointwise set-inclusion (with the assumption that $* \subseteq *$ and $\bot \subseteq x$ for every $x \in \mathcal{ER}_\tau$).*

Proposition 2. *Let $\tau \in Typing$. The map $\alpha_\tau^{\mathcal{ER}}$ is strict, additive and onto i.e., it is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{ER}_\tau$.*

We have explicitly calculated the optimal approximations induced by $\mathcal{ER}$ of all the concrete operations over states defined in [9]. We do not have space to fully describe them here. They are similar to those of the Palsberg and Schwartzbach's domain for *class analysis* [10] as formulated in [9]. However, $\mathcal{ER}$ observes the fields of just the reachable objects (Definition 15), while Palsberg and Schwartzbach observe the fields of all objects in memory. As an example, consider the operations $\mathsf{get\_var}^v$, which loads the value of the variable v in an accumulator called *res*, and $\mathsf{restrict}^{vs}$, which removes the variables in the set vs from the frame of the state. Their optimal approximations over $\mathcal{ER}$ are

$$\mathsf{get\_var}_\tau^v(\langle\phi,\mu\rangle) = \langle\phi[res \mapsto \phi(v)],\mu\rangle \qquad \mathsf{restrict}_\tau^{vs}(\langle\phi,\mu\rangle) = \xi_\tau(\langle\phi|_{-vs},\mu\rangle) \ .$$

In the case of $\mathsf{restrict}$, we remove the variables in vs from the abstract frame and, through ξ_τ, we force to $\varnothing$ the fields of type class of the objects which hence have no reachable creation point. Thus the result of $\mathsf{restrict}$ belongs to $\mathcal{ER}_{\tau|_{-vs}}$ (Definition 17).

We show now how an element $s \in \mathcal{E}$ can be *implemented* by an element of $\mathcal{ER}$. The idea is that every variable or field must be bound in $\mathcal{ER}$ to all those creation points in s compatible with its type.

Definition 18. *Let $\tau \in Typing$ and $s \subseteq \Pi$. We define $\vartheta_\tau(s) \in Frame_\tau^{\mathcal{ER}}$ such that, for every $v \in \mathsf{dom}(\tau)$,*

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

The implementation *$\theta_\tau(s) \in \mathcal{ER}_\tau$ of $s \in \mathcal{E}_\tau$ is $\theta_\tau(s) = \xi_\tau(\langle\vartheta_\tau(s), \vartheta_{\overline{\tau}}(s)\rangle)$.*

Example 14. Let τ be as in Figure 3 and $s = \{\overline{\pi}, \pi_1, \pi_2, \pi_3\} \in \mathcal{E}_\tau$ (Example 7). Then

$$\theta_\tau(s) = \langle \begin{bmatrix} \mathtt{f} \mapsto \{\pi_2, \pi_3\}, \mathtt{n} \mapsto \{\pi_2, \pi_3\} \\ \mathtt{out} \mapsto *, \mathtt{this} \mapsto \{\overline{\pi}\} \end{bmatrix}, \begin{bmatrix} \mathtt{next} \mapsto \{\pi_2, \pi_3\} \\ \mathtt{rotation} \mapsto \{\pi_1\}, \ldots \end{bmatrix} \rangle .$$

Proposition 3 proves the correctness of the implementation of Definition 18. It is based on the following result.

Lemma 5. *Let $\tau \in$ Typing, $\sigma \in \Sigma_\tau$ and $s \in \mathcal{E}_\tau$. We have $\theta_\tau(s) \in \mathcal{ER}_\tau$. Moreover, we have $\alpha_\tau^{\mathcal{E}}(\sigma) \subseteq s$ if and only if $\alpha_\tau^{\mathcal{ER}}(\sigma) \subseteq \theta_\tau(s)$.*

Proposition 3. *Let $\tau \in$ Typing, $s \in \mathcal{E}_\tau$ and $\gamma_\tau^{\mathcal{E}}$ and $\gamma_\tau^{\mathcal{ER}}$ be the concretisation maps induced by the abstraction maps of Definitions 9 and 15, respectively. We have $\gamma_\tau^{\mathcal{E}}(s) = \gamma_\tau^{\mathcal{ER}}(\theta_\tau(s))$ and hence $\gamma_\tau^{\mathcal{E}}(\mathcal{E}_\tau) \subseteq \gamma_\tau^{\mathcal{ER}}(\mathcal{ER}_\tau)$.*

Proof. By Lemma 5, for every $s \in \mathcal{E}_\tau$ we have [5]

$$\gamma_\tau^{\mathcal{E}}(s) = \{\sigma \in \Sigma_\tau \mid \alpha_\tau^{\mathcal{E}}(\sigma) \subseteq s\} = \{\sigma \in \Sigma_\tau \mid \alpha_\tau^{\mathcal{ER}}(\sigma) \subseteq \theta_\tau(s)\} = \gamma_\tau^{\mathcal{ER}}(\theta_\tau(s)).$$

The inclusion proved in Proposition 3 is strict, in general.

Example 15. Let τ be as in Figure 3. By Example 7 we have $\#\mathcal{E}_\tau \leq 2^4$ and $\#\mathcal{ER}_\tau > \#\mathit{Frame}_\tau^{\mathcal{ER}} \geq 2^4$.

7 Discussion

Abstract domains for escape analysis have been developed for both functional and object-oriented languages. For functional languages, the escape analysis first defined in [11] was later made more efficient [6] and then extended to some imperative constructs and applied to very large programs [2]. For object-oriented languages, there have been a number of approaches to escape analysis [3,4,7,8,12,15]. In [12], a *lifetime analysis* propagates the *sources* of data structures. In [3] integer *contexts* are used to specify the part of a data structure which can escape. Both [4] and [15] use *connection graphs* to represent the concrete memory although, in [15] these graphs are slightly more concrete than those used in [4]. In [7], a program is translated to a constraint, whose solution is the set of *escaping* variables. In [8], an escape analysis for object-oriented languages was defined on the *escape property* $\mathcal{E}$ and shown to be sometimes more precise than these other escape analyses.

In this paper, we have refined the escape property $\mathcal{E}$ defined in [8] to give the domain $\mathcal{ER}$ and derive the abstract analyser directly from the abstract domain. Observe that the relation between the refined domain $\mathcal{ER}$ and $\mathcal{E}$ is similar to that between the Palsberg and Schwartzbach's class analysis [9,10] and the *rapid type analysis* [1] although, while all objects stored in memory are considered in [1,9,10], only those actually reachable from some variable or field are considered by the domains $\mathcal{E}$ and $\mathcal{ER}$ (Definitions 9 and 15).

The escape analysis using the domain $\mathcal{ER}$ has been defined and implemented for the simple object-oriented language given in [14]. We are currently generalising the semantics to the Java bytecode. We will then be able to apply $\mathcal{ER}$ to the Java bytecode and to provide an experimental evaluation.

References

1. D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
2. B. Blanchet. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages (POPL'98)*, pages 25–37, San Diego, CA, USA, January 1998. ACM Press.
3. B. Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(1) of *SIGPLAN Notices*, pages 20–34, Denver, Colorado, USA, November 1999.
4. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(10) of *SIGPLAN Notices*, pages 1–19, Denver, Colorado, USA, November 1999.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
6. A. Deutsch. On the Complexity of Escape Analysis. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 358–371, Paris, France, January 1997. ACM Press.
7. D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In D. A. Watt, editor, *Compiler Construction, 9th International Conference, (CC'00)*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93, Berlin, Germany, March 2000. Springer-Verlag.
8. P. M. Hill and F. Spoto. A Foundation of Escape Analysis. Submitted for publication. Available from `http://www.sci.univr.it/~spoto/papers.html`, 2002.
9. T. Jensen and F. Spoto. Class Analysis of Object-Oriented Programs through Abstract Interpretation. In F. Honsell and M. Miculan, editors, *Proc. of FOSSACS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 261–275, Genova, Italy, April 2001. Springer-Verlag.
10. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.
11. Y. G. Park and B. Goldberg. Escape Analysis on Lists. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN Notices*, pages 116–127, San Francisco, California, USA, June 1992.
12. C. Ruggieri and T. P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *15th ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 285–293, San Diego, California, USA, January 1988.
13. F. Spoto. The LOOP Analyser. `http://www.sci.univr.it/~spoto/loop/`.
14. F. Spoto. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In P. Cousot, editor, *Proc. of the Static Analysis Symposium, SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 127–145, Paris, July 2001. Springer-Verlag.
15. J. Whaley and M. C. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(1) of *SIGPLAN Notices*, pages 187–206, Denver, Colorado, USA, November 1999.