A Foundation of Escape Analysis*

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 allows us to stack allocate dynamically created objects and to reduce the overhead of synchronisation in Java-like languages. We formalise the *escape property* $\mathcal{E}$, computed by an escape analysis, as an abstract interpretation of concrete states. We define the optimal abstract operations induced by $\mathcal{E}$ for a framework of analysis known as *watchpoint semantics*. The implementation of $\mathcal{E}$ inside that framework is a formally correct abstract semantics (analyser) for escape analysis. We claim that $\mathcal{E}$ is the basis for more refined and precise domains for escape analysis.

1 Introduction

Escape analysis has been studied for functional and for object-oriented languages. It allows us to stack allocate dynamically created data structures which would normally be heap allocated. This is possible if the data structures do not *escape* from the method which created them. It is well known that stack allocation reduces garbage collection overhead at run-time *w.r.t.* heap allocation. In the case of Java, escape analysis allows us to remove unnecessary synchronisations when an object is accessed. This is possible if the object does not *escape* the methods of its creating thread. This makes object accesses faster at run-time.

In this paper, we lay the foundations for the development of practical escape analysers for object-oriented languages. To do this, we first formalise the *escape property* $\mathcal{E}$ as a property (an *abstract interpretation*) of concrete states showing how it may interact with both the static type information and the late-binding mechanism. We show how, given a set of concrete operations, such as those given in [9,13] for executing a simple object-oriented language, optimal abstract operations induced by $\mathcal{E}$ may be constructed.

Note that, in this paper, we are concerned with providing a foundation for escape analysis and, as a consequence, the domain $\mathcal{E}$, which represents just the property of interest, does not include any other related information that could improve the precision of the analyser. Notwithstanding this, we show that, in

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

some cases, $\mathcal{E}$ with its abstract operations can be more precise than other proposed escape analyses. Moreover, our escape analyser can form a basis for more refined analysers, such as the one described in [8], with improved precision.

After a brief summary of our notation and terminology, in Section 3 we recall the watchpoint framework of [13] on which the analysis is based. Then, in Section 4, we formalise the escape property of interest and provide suitable abstract operations for its analysis while, in Section 5, we discuss our prototype implementation and experimental results. Section 6 concludes the paper by highlighting the differences between previous proposals and our own approach.

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 $\cdot$. A definition like $S = a \cdot b$, 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 4).

A *complete lattice* is a poset $C \cdot \leq$ where *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist. If $C \cdot \leq$ and $A \cdot \preceq$ are posets, then $f : C \mapsto A$ is *(co-)additive* if it preserves lub's (glb's). A map $f : A \mapsto A$ is a *lower closure operator* (*lco*) if it is *monotonic*, *reductive* and *idempotent*.

We recall now the basics of abstract interpretation (AI) [5]. Let $C \cdot \leq$ and $A \cdot \preceq$ 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. This is equivalent to α being onto, or γ one-to-one. If C and A are complete lattices and α is additive, it is the abstraction map of a Galois connection. 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* [5] works over *properties* of C *i.e.,* over $\wp(C)$ and is the fixpoint of the powerset extension of f. If f is defined through suboperations, their powerset extensions *and* $\cup$ (which merges the semantics of the branches of a conditional) induce the extension of f.

3 The Framework of Analysis

We build on the *watchpoint semantics* [13] which allows us to derive a compositional and *focused* analyser from a specification of a domain of abstract states

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


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


class square extends figure :
fields side, xcenter, ycenter : int
field rotation : angle
method def() : void is
  this.side := 1;
  this.xcenter := 0;
  this.ycenter := 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, xcenter, ycenter : int
method def() : void is
  this.radius := 1;   {w1}
  this.xcenter, this.ycenter := 0;
method draw() : void is
  % put something here...

class main :
method main() : void is
  f : figure;
  f := new square;     {π2}
  f.def();
  rotate(f);           {w2}
  f := new circle;     {π3}
  f.def();
  rotate(f);           {w3}
method rotate(f : figure) : void is
  a : angle;
  a := new angle;      {π4}
  f.rot(a);
  a.degree := 0;
  while (a.degree < 360)
    f.draw();
    a.degree := a.degree + 1;
```

Figure 1. An example of program.

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 watchpoint 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 [4,5,8].

The analyser presented here is for a simple typed object-oriented language where the concrete states and operations are based on [9]. However, here we require a more concrete notion of object. This is because, for escape analysis, every object has to be associated with its *creation point*.

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

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

Types and type environments are initialised as $\mathsf{init}(int) = 0$, $\mathsf{init}(\kappa) = nil$ for $\kappa \in \mathcal{K}$ and $\mathsf{init}(\tau)(v) = \mathsf{init}(\tau(v))$ for $\tau \in TypEnv$ and $v \in \mathsf{dom}(\tau)$.

A class contains local variables (*fields*) and functions (*methods*). A method has a set of input/output variables called *parameters*, including `out`, which holds

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

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

$$\begin{aligned}
F(\texttt{angle}) &= [\texttt{degree} \mapsto \mathit{int}] \qquad F(\texttt{figure}) = F(\texttt{main}) = [] \\
F(\texttt{square}) &= [\texttt{side} \mapsto \mathit{int}, \texttt{xcenter} \mapsto \mathit{int}, \texttt{ycenter} \mapsto \mathit{int}, \texttt{rotation} \mapsto \texttt{angle}] \\
F(\texttt{circle}) &= [\texttt{radius} \mapsto \mathit{int}, \texttt{xcenter} \mapsto \mathit{int}, \texttt{ycenter} \mapsto \mathit{int}] \\
M(\texttt{angle}) &= [\texttt{acute} \mapsto \texttt{angle_acute}] \\
M(\texttt{figure}) &= [\texttt{def} \mapsto \texttt{figure_def}, \texttt{rot} \mapsto \texttt{figure_rot}, \texttt{draw} \mapsto \texttt{figure_draw}] \\
M(\texttt{square}) &= [\texttt{def} \mapsto \texttt{square_def}, \texttt{rot} \mapsto \texttt{square_rot}, \texttt{draw} \mapsto \texttt{square_draw}] \\
M(\texttt{circle}) &= [\texttt{def} \mapsto \texttt{circle_def}, \texttt{rot} \mapsto \texttt{figure_rot}, \texttt{draw} \mapsto \texttt{circle_draw}] \\
M(\texttt{main}) &= [\texttt{main} \mapsto \texttt{main_main}, \texttt{rotate} \mapsto \texttt{main_rotate}] \\
P(\texttt{angle_acute}) &= [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{angle}] \\
P(\texttt{figure_rot}) &= [\texttt{a} \mapsto \texttt{angle}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{figure}] \\
P(\texttt{main_rotate}) &= [\texttt{f} \mapsto \texttt{figure}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{main}] \\
P(\texttt{figure_def}) &= [\texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{figure}] \text{ (the other cases of } P \text{ are like this)}
\end{aligned}$$

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

the result of the method, and `this`, which is the object over which the method has been called. *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 2. *Let $\mathcal{M}$ be a finite set of* methods. *We define*

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

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

Definition 3. *The* static information *of a program consists of a poset $\mathcal{K} \cdot \leq$, a set of methods $\mathcal{M}$ and maps $F \in \mathit{Fields}$, $M \in \mathit{Methods}$ and $P \in \mathit{Pars}$.*

An example program is given in Figure 1. Note that `void` methods are an abbreviation for methods which always return the integer 0, implicitly discarded by the caller. The static information of that program is shown in Figure 2.

Definition 4. *Let Π be a finite set of labels called* 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* $k(\overline{\pi}) = \mathtt{main}$*. Every other* $\pi \in \Pi$ *decorates a* `new` κ *statement in the program. Then we define* $k(\pi) = \kappa$*. Let* $\pi \in \Pi$*,* $F \in \mathit{Fields}$ *and* $M \in \mathit{Methods}$*. We define* $F(\pi) = F(k(\pi))$ *and* $M(\pi) = M(k(\pi))$*.*

We define a *frame* as a map that assigns values to variables. These *values* can be integers, locations or *nil* where a *location* is a memory cell. The value assigned to a variable must be consistent with its type. For instance, a class variable should be assigned to a location or to *nil*. A *memory* is a map from locations to objects where an *object* is characterized by its creation point and the frame of its fields. Figure 3 illustrates these different concepts. An *update* of a memory allows its frames to assign new (type consistent) values to the variables.

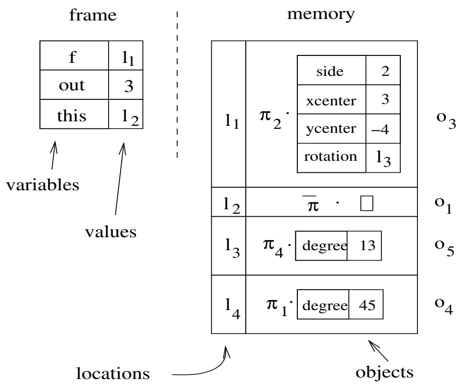

Figure 3. Frame ϕ_1 and memory μ_1 for type environment $\tau_{w_2} = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{main}]$.

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

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

$$\mathit{Obj} = \{\pi \cdot \phi \mid \pi \in \Pi,\ \phi \in \mathit{Frame}_{F(\pi)}\}$$

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

Let $\mu_1, \mu_2 \in \mathit{Memory}$ *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 1. Let $\tau_{w_2} = [\mathtt{f} \mapsto \mathtt{figure}, \mathtt{out} \mapsto \mathit{int}, \mathtt{this} \mapsto \mathtt{main}]$ be the type environment at program point w_2 in Figure 1. Letting $l_1, l_2 \in \mathit{Loc}$, the set $\mathit{Frame}_{\tau_{w_2}}$ contains ϕ_1 (see Figure 3) and ϕ_2 but it does not contain ϕ_3 and ϕ_4, where

$$\phi_1 = [\mathtt{f} \mapsto l_1, \mathtt{out} \mapsto 3, \mathtt{this} \mapsto l_2] \qquad \phi_2 = [\mathtt{f} \mapsto \mathit{nil}, \mathtt{out} \mapsto -2, \mathtt{this} \mapsto \mathit{nil}]$$

$$\phi_3 = [\mathtt{f} \mapsto 2, \mathtt{out} \mapsto -2, \mathtt{this} \mapsto l_1] \qquad \phi_4 = [\mathtt{f} \mapsto l_1, \mathtt{out} \mapsto 3, \mathtt{this} \mapsto 3]\ .$$

This is because `f` is bound to 2 in ϕ_3 (while it has class `figure` in τ_{w_2}) and `this` is bound to 3 in ϕ_4 (while it has class `main` in τ_{w_2}).

Example 2. Consider the program and its static information given in Figures 1 and 2. Since $F(\overline{\pi}) = F(\mathtt{main}) = []$, the only object created at $\overline{\pi}$ is $o_1 = \overline{\pi} \cdot []$. Objects created at π_2 have class $k(\pi_2) = \mathtt{square}$. Examples of such objects (where $l_3 \in \mathit{Loc}$), that are consistent with the definition of $F(\mathtt{square})$, are

$$o_2 = \pi_2 \cdot [\mathtt{side} \mapsto 4, \mathtt{xcenter} \mapsto 3, \mathtt{ycenter} \mapsto -5, \mathtt{rotation} \mapsto \mathit{nil}]\ ,$$

$$o_3 = \pi_2 \cdot [\mathtt{side} \mapsto 2, \mathtt{xcenter} \mapsto 3, \mathtt{ycenter} \mapsto -4, \mathtt{rotation} \mapsto l_3]\ .$$

Finally, examples of objects created in π_1 and π_4, respectively, are

$$o_4 = \pi_1 \cdot [\mathtt{degree} \mapsto 45] \qquad o_5 = \pi_4 \cdot [\mathtt{degree} \mapsto 13] .$$

(o_1, o_3, o_4 and o_5 are illustrated in Figure 3.)

Example 3. Consider locations $l_1, l_2, l_3, l_4 \in \mathit{Loc}$ and the objects o_1, o_2, o_3, o_4, o_5 from Example 2. Then *Memory* contains $\mu_1 = [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4]$ (Figure 3), $\mu_2 = [l_1 \mapsto o_1, l_2 \mapsto o_1]$ and $\mu_3 = [l_1 \mapsto o_2, l_2 \mapsto o_3, l_3 \mapsto o_1]$.

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

Definition 6. *Let $\tau \in \mathit{TypEnv}$, $\phi \in \mathit{Frame}_\tau$ 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 $k((\mu\phi(v)).\pi) \le \tau(v)$.*

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 $\tau \in \mathit{TypEnv}$, $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is* τ-correct *w.r.t. μ, and we write $\phi \cdot \mu : \tau$, 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 $F(o.\kappa)$-correct w.r.t. μ.*

Example 4. Let τ_{w_2}, ϕ_1 and ϕ_2 from Example 1, μ_1, μ_2 and μ_3 from Example 3. Let $\tau = \tau_{w_2}$. We have

- $\phi_1 \cdot \mu_1 : \tau$ (Figure 3). Condition 1 of Definition 7 holds because $\{v \in \mathsf{dom}(\phi_1) \mid \phi_1(v) \in \mathit{Loc}\} = \{\mathtt{this}, \mathtt{f}\}$, $\{l_1, l_2\} \subseteq \mathsf{dom}(\mu_1)$, $k(\mu_1(l_1).\pi) = k(\pi_2) = \mathtt{square} \le \mathtt{figure} = \tau(\mathtt{f})$ and $k(\mu_1(l_2).\pi) = k(\overline{\pi}) = \mathtt{main} = \tau(\mathtt{this})$. Condition 2 holds because the only $o \in \mathsf{rng}(\mu_1)$ such that $\mathsf{rng}(o.\phi) \cap \mathit{Loc} \neq \varnothing$ is o_3, since $\{l_3\} = \mathsf{rng}(o_3.\phi) \cap \mathit{Loc}$. Moreover, $k(\mu_1(l_3).\pi) = k(\pi_4) = \mathtt{angle} = F(o_3.\pi)(\mathtt{rotation})$.
- $\phi_2 \cdot \mu_2 : \tau$. Condition 1 of Definition 7 holds since there is no $v \in \mathsf{dom}(\phi_2)$ such that $\phi_2(v) \in \mathit{Loc}$. Condition 2 holds since $\mathsf{rng}(\mu_2) = \{o_1\}$ and $o_1.\phi = []$.
- $\phi_1 \cdot \mu_2 : \tau$ is false since condition 1 of Definition 7 does not hold. Namely, $\tau(\mathtt{f}) = \mathtt{figure}$, $k((\mu_2\phi_1(\mathtt{f})).\pi) = k(o_1.\pi) = k(\overline{\pi}) = \mathtt{main}$ and $\mathtt{main} \not\le \mathtt{figure}$.
- $\phi_2 \cdot \mu_3 : \tau$ is false since condition 2 of Definition 7 does not hold. Namely, $o_3 \in \mathsf{rng}(\mu_3)$ is such that $o_3.\phi$ is not weakly $F(o_3.\pi)$-correct *w.r.t.* μ_3, since $o_3.\phi(\mathtt{rotation}) = l_3$, $\mathtt{main} \not\le \mathtt{angle}$ but $k(\mu_3(l_3).\pi) = k(o_1.\pi) = k(\overline{\pi}) = \mathtt{main}$ and we have $F(o_3.\pi)(\mathtt{rotation}) = F(\mathtt{square})(\mathtt{rotation}) = \mathtt{angle}$.

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 $\tau \in \mathit{TypEnv}$. We define the* states

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

and the operations over states in Figure 4 (for their signature, see [13]).

$$\mathsf{nop}_\tau(\phi\cdot\mu)=\phi\cdot\mu \qquad \mathsf{get\_int}^i_\tau(\phi\cdot\mu)=\phi[res\mapsto i]\cdot\mu,\ i\in\mathbb{Z}$$
$$\mathsf{get\_nil}^\kappa_\tau(\phi\cdot\mu)=\phi[res\mapsto nil]\cdot\mu,\ \kappa\in\mathcal{K} \qquad \mathsf{get\_var}^v_\tau(\phi\cdot\mu)=\phi[res\mapsto\phi(v)]\cdot\mu,\ v\in\mathsf{dom}(\tau)$$
$$\mathsf{restrict}^{vs}_\tau(\phi\cdot\mu)=\phi|_{-vs}\cdot\mu,\ vs\subseteq\mathsf{dom}(\tau) \qquad \mathsf{expand}^{v:t}_\tau(\phi\cdot\mu)=\phi[v\mapsto\mathsf{init}(t)]\cdot\mu,\ t\in Type$$
$$\mathsf{put\_var}^v_\tau(\phi\cdot\mu)=\phi[v\mapsto\phi(res)]|_{-res}\cdot\mu,\ v\in\mathsf{dom}(\tau)$$
$$\mathsf{get\_field}^f_\tau(\phi'\cdot\mu)=\begin{cases}\phi'[res\mapsto((\mu\phi'(res)).\phi)(f)]\cdot\mu & \text{if } \phi'(res)\neq nil\\ \text{undefined} & \text{otherwise}\end{cases}$$
$$\mathsf{put\_field}^f_{\tau,\tau'}(\phi_1\cdot\mu_1)(\phi_2\cdot\mu_2)=\begin{cases}\phi_2|_{-res}\cdot\mu_2[l\mapsto\mu_2(l).\pi\cdot\mu_2(l).\phi[f\mapsto\phi_2(res)]]\\ \qquad \text{if } (l=\phi_1(res))\neq nil \text{ and } \mu_1\lhd_l\mu_2\\ \text{undefined} \qquad \text{otherwise}\end{cases}$$
$$=_\tau(\phi_1\cdot\mu_1)(\phi_2\cdot\mu_2)=\begin{cases}\phi_2[res\mapsto 1]\cdot\mu_2 & \text{if } \phi_1(res)=\phi_2(res)\\ \phi_2[res\mapsto -1]\cdot\mu_2 & \text{if } \phi_1(res)\neq\phi_2(res)\end{cases}$$
$$+_\tau(\phi_1\cdot\mu_1)(\phi_2\cdot\mu_2)=\phi_2[res\mapsto\phi_1(res)+\phi_2(res)]\cdot\mu_2$$
$$\mathsf{is\_nil}_\tau(\phi\cdot\mu)=\begin{cases}\phi[res\mapsto 1]\cdot\mu & \text{if } \phi(res)=nil\\ \phi[res\mapsto -1]\cdot\mu & \text{otherwise}\end{cases}$$
$$\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(\phi\cdot\mu)=[\iota_1\mapsto\phi(v_1),\ldots,\iota_n\mapsto\phi(v_n),\mathtt{this}\mapsto\phi(res)]\cdot\mu$$
$$\text{where } \{\iota_1,\ldots,\iota_n\}=P(\nu)\setminus\{\mathtt{out},\mathtt{this}\} \text{ (alphabetically ordered) and } \nu\in\mathcal{M}$$
$$\mathsf{return}^\nu_\tau(\phi_1\cdot\mu_1)(\phi_2\cdot\mu_2)=\begin{cases}\phi_1[res\mapsto\phi_2(\mathtt{out})]\cdot\mu_2 & \text{if } \mu_1\lhd_{\mathsf{rng}(\phi_1)|_{-res}\cap Loc}\mu_2\\ \text{undefined} & \text{otherwise}\end{cases}$$
$$\mathsf{new}^\pi_\tau(\phi\cdot\mu)=\phi[res\mapsto l]\cdot\mu[l\mapsto\pi\cdot\mathsf{init}(F(\pi))],\ \pi\in\Pi,\ l\in Loc\setminus\mathsf{dom}(\mu)$$
$$\mathsf{lookup}^{m,\nu}_\tau(\phi\cdot\mu) \text{ iff } \phi(res)\neq nil \text{ and } M((\mu\phi(res)).\pi)(m)=\nu$$
$$\mathsf{is\_true}_\tau(\phi\cdot\mu) \text{ iff } \phi(res)\geq 0 \qquad \mathsf{is\_false}_\tau(\phi\cdot\mu) \text{ iff } \phi(res)<0\ .$$

Figure 4. The operations over concrete states.

Example 5. In the hypotheses of Example 4, we have $\phi_1\cdot\mu_1\in\Sigma_\tau$ (Figure 3), $\phi_2\cdot\mu_2\notin\Sigma_\tau$ although $\phi_2\cdot\mu_2:\tau$ (Example 4), since $\mathtt{this}\in\mathsf{dom}(\tau)$ and $\phi_2(\mathtt{this})=nil$. We have $\phi_1\cdot\mu_2\notin\Sigma_\tau$ and $\phi_2\cdot\mu_3\notin\Sigma_\tau$ (Example 4).

In Figure 4, the subscript τ constrains the signature of the operations (see [13]). The variable *res* holds the intermediate results, like the top element of the operand stack of the Java virtual machine. The nop operation does nothing. The get operations load a constant, the value of another variable or of the field of an object in *res*. In the last case, that object is assumed to be stored in *res before* the get operation. 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 5). For every binary operation over values like $=$ and $+$, there is an operation on states. Note (in the case of $=$) that Booleans are implemented through integers (every non-negative integer means true). The operation is_nil checks whether *res* points to *nil*. 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 5) requires that the variables of σ have not been changed during the execution of the method. The operation expand (restrict) adds (removes) variables. The operation new^π creates a new object of creation point π. The predicate $\mathsf{lookup}^{m,\nu}$ checks if by calling the method identified by m of the object pointed to by *res*, the method ν is run. The predicate $\mathsf{is\_true}$ ($\mathsf{is\_false}$) checks if *res* contains true (false).

Example 6. Let τ_{w_2} and ϕ_1 as in Example 1 and μ_1 as in Example 3 (see also Figure 3). The state $\sigma_1 = \phi_1 \cdot \mu_1$ (Example 5) could be the current state at program point w_2 in Figure 1. The computation continues as follows [13]. Let $\tau = \tau_{w_2}$ and $\tau' = \tau[res \mapsto \texttt{figure}]$. The sequence of operations which are executed is

$$\begin{array}{rll}
\sigma_2 = \mathsf{new}_\tau^{\pi_3}(\sigma_1) & \text{create an object of class } \texttt{circle} & \\
\sigma_3 = \mathsf{put_var}^{\texttt{f}}_{\tau[res\mapsto\texttt{circle}]}(\sigma_2) & \text{assign it to } \texttt{f} & \\
\sigma_4 = \mathsf{get_var}^{\texttt{f}}_\tau(\sigma_3) & \text{read } \texttt{f} & \\
\mathsf{lookup}^{\texttt{def},\texttt{figure_def}}_{\tau'}(\sigma_4) & \text{if it is true, let } \nu = \texttt{figure_def} & (1)\\
\mathsf{lookup}^{\texttt{def},\texttt{square_def}}_{\tau'}(\sigma_4) & \text{if it is true, let } \nu = \texttt{square_def} & \\
\mathsf{lookup}^{\texttt{def},\texttt{circle_def}}_{\tau'}(\sigma_4) & \text{if it is true, let } \nu = \texttt{circle_def} & \\
\sigma_5 = \mathsf{call}^{\nu}_{\tau[res\mapsto P(\nu)(\texttt{this})]}(\sigma_4) & \text{initialise the selected method } \nu. &
\end{array}$$

Let $o_6 = \pi_3 \cdot [\texttt{radius} \mapsto 0, \texttt{xcenter} \mapsto 0, \texttt{ycenter} \mapsto 0]$ and $l \in Loc \setminus \{l_1, l_2, l_3, l_4\}$. We have

$$\sigma_2 = [\texttt{f} \mapsto l_1, \texttt{out} \mapsto 3, res \mapsto l, \texttt{this} \mapsto l_2] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6]$$
$$\sigma_3 = [\texttt{f} \mapsto l, \texttt{out} \mapsto 3, \texttt{this} \mapsto l_2] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6]$$
$$\sigma_4 = [\texttt{f} \mapsto l, \texttt{out} \mapsto 3, res \mapsto l, \texttt{this} \mapsto l_2] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6] \ .$$

Consider the lookup checks. They determine which is the target of the virtual call $\texttt{f.def}()$ in Figure 1. We have $(\sigma_4.\phi)(res) = l \neq nil$ and $(\sigma_4.\mu)(l) = o_6$. Then the method of o_6 identified by $\texttt{def}$ is called. We have $o_6.\pi = \pi_3$ and $k(\pi_3) = \texttt{circle}$. Moreover $M(\texttt{circle})(\texttt{def}) = \texttt{circle\_def}$ (Figure 2). Then the only lookup check which succeeds is the last one, $\nu = \texttt{circle\_def}$ is called and

$$\sigma_5 = [\texttt{this} \mapsto l] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6] \ .$$

Note that the object o_6 created by the $\texttt{new}$ statement is now the $\texttt{this}$ object of this instantiation of the method $\texttt{circle\_def}$.

In [13] it is shown that every abstraction of $\wp(\Sigma_\tau)$ and of the powerset extension of the operations in Figure 4 and of $\cup$ induces an abstraction of the watchpoint semantics. We will use this result in the next section.

4 The Property $\mathcal{E}$

"Escape analysis determines whether the lifetime of data exceeds its static scope" [2]. "An object o is said to escape a method m if the lifetime of o may exceed the lifetime of m" [3]. "Escape analysis computes bounds of where references to newly created objects may occur" [7]. These definitions of escape analysis for object-oriented languages are all unsatisfactory for a number of reasons. First of all, it is not clear which is the *actor* of the analysis (data, objects, references?). Furthermore, they use informal concepts (*lifetime*, *bound*, *newly created*). Finally, it is not clear whether escape analysis is a definite or possible analysis (do we want a subset or a superset of the "data whose lifetime exceeds their static scope"?). Thus a comparison of the precision of different analyses is possible only by an experimental evaluation where the number and degree of optimisations they achieve are compared. However, even this would be difficult as a standard and representative set of benchmarks is needed. In addition to this, the definitions refer to properties of programs rather than properties of states. For instance, *lifetime* refers implicitly to different moments in the execution of a program. So that, although the above definitions are useful for understanding the problem, they cannot be used to define abstract interpretations of states.

We now want to define more formally the *escape property* $\mathcal{E}$ as a property of states, independently of the optimisations it allows. This is important because:

- It makes clear that escape analysis is a *possible* analysis of creation points,
- $\mathcal{E}$ is the *simplest* abstract domain for escape analysis which is a concretisation of the escape property itself,
- As a property of states, it can be defined without any consideration about name clashes or multiple instances of the same variable during recursion,
- It allows a *formal* comparison of different domains *w.r.t.* their precision. It is indeed enough to compute and compare their *quotients w.r.t.* $\mathcal{E}$ [4,8].

Here is our definition of escape analysis, which we will further formalise later.

Escape analysis *over*approximates, for every program point p, the set of creation points of objects reachable in p from some variable or field in scope.

The choice of an *over*approximation follows from the typical use of the information provided by *escape analysis*. For instance, an object can be stack allocated [2,3,7] 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. For the same reason, we are interested in the creation points of the objects and not in their identity.

Example 7. Consider the program in Figure 1. Are there any objects created in π_4 and reachable in program point w_2? We have $\tau_{w_2} = [\texttt{f} \mapsto \texttt{figure}, \texttt{out} \mapsto \mathit{int}, \texttt{this} \mapsto \texttt{main}]$ (Example 1). We cannot reach any object from `out`, because it can only contain integers. The variable `this` has class `main` which has no fields. Since in π_4 we create objects of class `angle`, they cannot be reached from `this`. The variable `f` has class `figure`, which has no fields. Reasoning as for `this`, we could conclude that no object created in π_4 can be reached from `f`. That conclusion is wrong. Indeed, since `f` contains a `square`, the call `rotate(f)` results in a call `f.rot(a)` which stores `a`, created in π_4, in the field `rotation`, and that field can later be accessed (for instance, by `f.draw()`).

The considerations in Example 7 lead to the definition of *reachability*, where we use the actual fields of the objects instead of those of its declared class. We use the following result to guarantee that Definition 9 is well-given.

Lemma 1. *Let* $\tau \in \mathit{TypEnv}$, $\phi \cdot \mu \in \Sigma_\tau$ *and* $o \in \mathsf{rng}(\mu)$. *Then* $o.\phi \cdot \mu \in \Sigma_{F(o.\pi)}$.

Definition 9. *Let* $\tau \in$ *TypEnv and* $\sigma = \phi \cdot \mu \in \Sigma_\tau$*. The set of the creation points of the objects* reachable *in* σ *is* $\alpha_\tau(\sigma) = \cup\{\alpha^i_\tau(\sigma) \mid i \geq 0\} \subseteq \Pi$*, where*

$$\alpha^0_\tau(\sigma) = \varnothing$$
$$\alpha^{i+1}_\tau(\sigma) = \cup\{\{o.\pi\} \cup \alpha^i_{F(o.\pi)}(o.\phi \cdot \mu) \mid v \in \mathsf{dom}(\phi), \phi(v) \in Loc,\ o = \mu\phi(v)\}.$$

The maps α_τ *and* α^i_τ *are pointwise extended to* $\wp(\Sigma_\tau)$*.*

Variables and fields of type *int* do not contribute to α_τ. Moreover, since Π is finite and $\alpha^i_\tau \subseteq \alpha^{i+1}_\tau$ (by induction over i), for every $S \subseteq \Pi$ there is $k \in \mathbb{N}$ such that $\alpha_\tau(S) = \alpha^k_\tau(S)$.

Example 8. In the hypotheses of Figure 3, we have $\alpha_\tau(\phi_1 \cdot \mu_1) = \{\overline{\pi}, \pi_2, \pi_4\}$. Note that o_4 is not reachable.

In general, it can be $\mathsf{rng}(\alpha_\tau) \neq \wp(\Pi)$, since α_τ is not necessarily onto.

Example 9. In the hypotheses of Figure 2, let $\tau = [\mathtt{a} \mapsto \mathtt{circle}, \mathtt{this} \mapsto \mathtt{main}]$ and $\sigma = \phi \cdot \mu \in \Sigma_\tau$. Then $\pi_4 \notin \alpha_\tau(\sigma)$, since for every $i \in \mathbb{N}$ we have

$$\alpha^{i+1}_\tau(\sigma) = \cup\{\{o.\pi\} \cup \alpha^i_{F(o.\pi)}(o.\phi \cdot \mu) \mid v \in \{\mathtt{a}, \mathtt{this}\},\ \phi(v) \in Loc,\ o = \mu\phi(v)\}. \quad (2)$$

Since $k(o.\pi) \in \{\mathtt{circle}, \mathtt{main}\}$ has no class fields, (2) is equal to

$$\{o.\pi \mid v \in \{\mathtt{a}, \mathtt{this}\},\ \phi(v) \in Loc,\ o = \mu\phi(v)\}\ . \quad (3)$$

Since π_1, π_2 and π_4 create objects incompatible with `circle` and `main`, (3) is contained in $\{\overline{\pi}, \pi_3\}$. Since i is arbitrary, we have the thesis.

Example 9 shows that *static type information provides escape information*, by indicating some creation points which create objects that cannot be reached.

Let $S \in \wp(\Pi)$. We define $\delta_\tau(S)$ as the largest subset of S which contains only those creation points deemed useful by the type environment τ. Note that if there are no possible creation points for `this`, all creation points are useless.

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

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

In Definition 10 we consider all subclasses of κ (Example 7). We have $\delta_\tau = \delta^{\#\Pi}_\tau$.

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

Example 10. Program point w_1 in Figure 1 has type environment $\tau_{w_1} = [\mathtt{out} \mapsto int, \mathtt{this} \mapsto \mathtt{circle}]$. Let $S = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$. For every $i \in \mathbb{N}$ we have

$$\begin{aligned}\delta^{i+1}_{\tau_{w_1}}(S) &= \cup\{\{\pi\} \cup \delta^i_{F(\pi)}(S) \mid \kappa \in \mathsf{rng}(\tau_{w_1}) \cap \mathcal{K},\ \pi \in S,\ k(\pi) \leq \kappa\} \\ &= \cup\{\{\pi\} \cup \delta^i_{F(\pi)}(S) \mid \pi \in S,\ k(\pi) \leq \mathtt{circle}\} = \{\pi_3\} \cup \delta^i_{F(\mathtt{circle})}(S)\ .\end{aligned}$$

Since $\mathsf{rng}(F(\mathtt{circle})) = \{int\}$, we conclude that $\delta_{\tau_{w_1}}(S) = \{\pi_3\}$.

Example 11. Consider the program point w_2 in Figure 1 and its type environment τ_{w_2} from Example 1. Let $S = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$. For every $i \in \mathbb{N}$ we have

$$\begin{aligned}
\delta^{i+2}_{\tau_{w_2}}(S) &= \cup\{\{\pi\} \cup \delta^{i+1}_{F(\pi)}(S) \mid \kappa \in \{\texttt{figure},\texttt{main}\},\ \pi \in S,\ k(\pi) \le \kappa\} \\
&= \{\overline{\pi}, \pi_2, \pi_3\} \cup \delta^{i+1}_{F(\texttt{square})}(S) \cup \delta^{i+1}_{F(\texttt{circle})}(S) \cup \delta^{i+1}_{F(\texttt{main})}(S) \\
&= \{\overline{\pi}, \pi_2, \pi_3\} \cup \delta^{i+1}_{F(\texttt{square})}(S) \\
&= \{\overline{\pi}, \pi_2, \pi_3\} \cup (\cup\{\{\pi\} \cup \delta^{i}_{F(\pi)}(S) \mid \pi \in S,\ k(\pi) \le \texttt{angle}\}) \\
&= \{\overline{\pi}, \pi_2, \pi_3\} \cup (\{\pi_1, \pi_4\} \cup \delta^{i}_{F(\texttt{angle})}(S)) = \{\overline{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}\ .
\end{aligned}$$

Then all creation points are useful in w_2 (compare this with Example 7).

The following result proves that δ_τ can be used to define $\mathsf{rng}(\alpha_\tau)$.

Lemma 2. *Let $\tau \in$ TypEnv. Then $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha_\tau)$ and $\varnothing \in \mathsf{fp}(\delta_\tau)$. If, in addition,* `this` $\in \mathsf{dom}(\tau)$*, then for every $X \subseteq \Sigma_\tau$ we have $\alpha_\tau(X) = \varnothing$ if and only if $X = \varnothing$.*

Lemma 2 allows us to assume that $\alpha_\tau : \wp(\Sigma_\tau) \mapsto \mathsf{fp}(\delta_\tau)$. Moreover, it justifies the following definition of the simplest domain $\mathcal{E}$ for escape analysis. It coincides with the *escape property* itself. Therefore, Lemma 2 can be used to compute the possible values of the escape property at a given program point. However, it does not specify which of these is best. This is the goal of an escape analysis.

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

Example 12. Let τ_{w_1} and τ_{w_2} be as in Examples 10 and 1, respectively. We have

$$\mathcal{E}_{\tau_{w_1}} = \{\varnothing, \{\pi_3\}\}\ , \quad \mathcal{E}_{\tau_{w_2}} = \{\varnothing\} \cup \left\{ S \in \wp(\Pi) \,\middle|\, \begin{array}{l} \overline{\pi} \in S \text{ and} \\ \pi_1 \in S \text{ or } \pi_4 \in S \text{ entails } \pi_2 \in S \end{array} \right\}.$$

The constraint on $\mathcal{E}_{\tau_{w_2}}$ says that to reach an `angle` (created in π_1 or in π_4) from the variables in $\mathsf{dom}(\tau_{w_2})$, we must be able to reach a `square` (created in π_2).

By Definition 9, if $\tau \in$ *TypEnv*, then α_τ is strict and additive and, by Lemma 2, α_τ is onto $\mathcal{E}_\tau$. Thus we have the following result.

Proposition 2. *Let $\tau \in$ TypEnv. The map α_τ (Definition 9) is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to $\mathcal{E}_\tau$.*

Note that if, in Definition 11, we had defined $\mathcal{E}_\tau$ as $\wp(\Pi)$, the map α_τ would induce just a Galois connection instead of a Galois insertion, as a consequence of Lemma 2.

The domain $\mathcal{E}$ of Definition 11 can be used to compare abstract domains *w.r.t.* the *escape property* [4]. Moreover, since it is an abstract domain, it induces optimal abstract operations which can be used for an actual escape analysis.

Proposition 3. *The operations in Figure 5 are the optimal counterparts induced by α of the operations in Figure 4 and of $\cup$. They are implicitly strict on $\varnothing$, except for* return*, which is strict in its first argument only, and for $\cup$.*

$$
\begin{aligned}
\mathsf{nop}_\tau(S) &= S & \mathsf{get\_int}^i_\tau(S) &= S\\
\mathsf{get\_nil}^\kappa_\tau(S) &= S & \mathsf{get\_var}^v_\tau(S) &= S\\
\mathsf{is\_true}_\tau(S) &= S & \mathsf{is\_false}_\tau(S) &= S\\
\mathsf{put\_var}^v_\tau(S) &= \delta_{\tau|_{-v}}(S) & \mathsf{is\_nil}_\tau(S) &= \delta_{\tau|_{-res}}(S)\\
\mathsf{new}^\pi_\tau(S) &= S \cup \{\pi\} & =_\tau(S_1)(S_2) = +_\tau(S_1)(S_2) &= S_2\\
\mathsf{expand}^{v:t}_\tau(S) &= S & \mathsf{restrict}^{vs}_\tau(S) &= \delta_{\tau_{-vs}}(S)\\
\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau(S) &= \delta_{\tau|_{\{v_1,\ldots v_n,res\}}}(S) & \cup_\tau(S_1)(S_2) &= S_1 \cup S_2
\end{aligned}
$$

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

$$
\mathsf{put\_field}^f_{\tau,\tau'}(S_1)(S_2) = \begin{cases} \varnothing & \text{if } \{\pi \in S_1 \mid k(\pi) \le \tau(res)\} = \varnothing\\ \delta_{\tau|_{-res}}(S_2) & \text{otherwise}\end{cases}
$$

$$
\mathsf{return}^\nu_\tau(S_1)(S_2) = \cup\{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K},\ \pi \in S_1,\ k(\pi) \le \kappa\} \cup S_2
$$

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

Figure 5. The optimal abstract operations over $\mathcal{E}$.

Note how the operations in Figure 5 make extensive use through δ of the static type information of the variables to improve the precision of the analysis.

We provide now the worst-case complexity of the analysis induced by $\mathcal{E}$.

Proposition 4. *Let $p = \#\Pi$, n be the number of commands of the program P to be analysed, b the number of nodes of the call graph of P and t the maximal number of fields of an object or of variables in scope. The cost in time of the analysis induced by $\mathcal{E}$ is $O(bnt2^p)$.*

Example 13. Let us mimic, in $\mathcal{E}$, the concrete computation (1) of Example 6. We start from the abstraction (Definition 9) of σ_1 *i.e.,* $\{\overline{\pi}, \pi_2, \pi_4\}$ (Example 8).

$$
\begin{aligned}
S_1 &= \alpha_\tau(\{\sigma_1\}) = \alpha_\tau(\sigma_1) = \{\overline{\pi}, \pi_2, \pi_4\}\\
S_2 &= \mathsf{new}^{\pi_3}_\tau(S_1) = \{\overline{\pi}, \pi_2, \pi_3, \pi_4\}\\
S_3 &= \mathsf{put\_var}^{\mathtt{f}}_{\tau[res \mapsto \mathtt{circle}]}(S_2) = \delta_{\left[\substack{\mathtt{out} \mapsto int, res \mapsto \mathtt{circle}\\ \mathtt{this} \mapsto \mathtt{main}}\right]}(\{\overline{\pi}, \pi_2, \pi_3 . \pi_4\}) = \{\overline{\pi}, \pi_3\}\\
S_4 &= \mathsf{get\_var}^{\mathtt{f}}_\tau(S_3) = \{\overline{\pi}, \pi_3\}\ .
\end{aligned}
$$

Note that $\pi_2 \notin S_3$ *i.e.,* the analysis induced by $\mathcal{E}$ is far from being a blind *accumulation* of creation points. Moreover, that information allows us to guess precisely the target of the virtual call `f.def()`. Indeed, the abstract lookup computes the abstract states S', S'' and S''' which hold if its target is `figure_def`, square_def and circle_def, respectively, and only one of them is non-empty.

$$S' = \mathsf{lookup}^{\mathtt{def},\mathtt{figure\_def}}_{\tau'}(S_4) = \varnothing$$
$$\text{since } \{\pi \in S_4 \mid k(\pi) \leq \mathtt{figure} \text{ and } M(\pi)(\mathtt{def}) = \mathtt{figure\_def}\} = \varnothing$$
$$S'' = \mathsf{lookup}^{\mathtt{def},\mathtt{square\_def}}_{\tau'}(S_4) = \varnothing$$
$$\text{since } \{\pi \in S_4 \mid k(\pi) \leq \mathtt{figure} \text{ and } M(\pi)(\mathtt{def}) = \mathtt{square\_def}\} = \varnothing$$
$$S''' = \mathsf{lookup}^{\mathtt{def},\mathtt{circle\_def}}_{\tau'}(S_4)$$
$$= \delta_\tau(S_4) \cup \left(\bigcup \left\{ \{\pi\} \cup \delta_{F(\pi)}(S_4) \,\middle|\, \pi \in S \right\}\right) = \{\overline{\pi}, \pi_3\} \cup \{\pi_3\} = \{\overline{\pi}, \pi_3\} ,$$
$$\text{since } S = \{\pi \in S_4 \mid k(\pi) \leq \mathtt{figure},\ M(\pi)(\mathtt{def}) = \mathtt{circle\_def}\} = \{\pi_3\}.$$

The abstract operations over $\mathcal{E}$ are strict on $\varnothing$ (Proposition 3). Thus only S''' contributes non-trivially to the static analysis. Namely, the starting state of the selected target method circle_def is

$$S_5 = \mathsf{call}^{\mathtt{circle\_def}}_{\tau[res\mapsto\mathtt{circle}]}(S''') = \delta_{[res\mapsto\mathtt{circle}]}(S''') = \{\pi_3\} .$$

Note that S_i is the *exact* abstraction (Definition 9) of the state σ_i of Example 6 for $i = 1, \ldots, 5$.

5 Implementation

We have implemented the domain $\mathcal{E}$ for escape analysis of Section 4 inside a static analyser for simple object-oriented programs, called LOOP [12,13].

Consider the program in Figure 1. The type environment of the input of main is $\tau_{in} = [\mathtt{this} \mapsto \mathtt{main}]$, that of its output is $\tau_{out} = [\mathtt{out} \mapsto int]$. The abstract domain for τ_{in} is $\mathcal{E}_{\tau_{in}} = \{\varnothing, \{\overline{\pi}\}\}$ (Definition 11). LOOP, focused on the watchpoints w_2 and w_3, concludes that if the input of main is $\varnothing$, which is the abstraction of the empty set of concrete states (Lemma 2), its output is $\varnothing$ and no watchpoints are observed. If the input is $\overline{\pi}$, the output is still $\varnothing$ (no objects can be reached from an integer) *but* the watchpoints w_2 and w_3 are observed. They have the same type environment (Example 1). LOOP concludes that, in w_2, only objects created in $\{\overline{\pi}, \pi_1, \pi_2, \pi_4\}$ can be reached. Since the creation point π_4 is internal to the method main_rotate, this means, in particular, that the new angle in π_4 cannot be stack allocated *for this instantiation of* main_rotate. Instead, LOOP concludes that in w_3 only objects created in $\{\overline{\pi}, \pi_3\}$ are reachable. This time, the new angle in π_4 can be stack allocated.

The fact that no objects created in π_4 are reachable in w_3 follows from two properties of our analysis. The first is that it is context-sensitive *i.e.,* it allows us to distinguish what happens in w_2 and in w_3. The second is that it uses *optimal* abstract operations (Figure 5) which, in particular, can sometimes restrict the target of a virtual call. In our example, since we know that f is bound to a circle after π_3, we conclude that the subsequent call rotate(f) results in a virtual call f.rot(a) which resolves into the method circle_rot. That method does not use a. Thus, the object stored in a does not escape the rotate(f) call.

	[2]	[3]	[7]	[11]	[14]	$\mathcal{E}$
Is the analysis context-sensitive?	Y	Y	N	Y	Y	Y
Is the analysis compositional?	Y	N	Y	Y	Y	Y
Is there a correctness proof?	Y	N	N	N	N	Y
Is there an optimality proof of the operations?	N	N	N	N	N	Y
Is the access to a field somehow approximated?	Y	Y	N	Y	Y	Y
Are virtual calls somehow restricted?	N	N	N	N	N	Y
Can it derive that π_4 is not reachable in w_3 (Figure 1) ?	N	N	N	N	N	Y

Figure 6. A comparison of different escape analyses.

6 Discussion

The first escape analysis for functional languages [10] has been made more efficient in [6] and then extended to some imperative constructs and applied to very large programs [1]. For object-oriented languages, escape analysis has been considered in [2,3,7,8,11,14]. In [11], a *lifetime analysis* propagates the *sources* of data structures. In [2] integer *contexts* are used to specify the part of a data structure which can escape. Both [3] and [14] use *connection graphs* to represent the concrete memory, but in [14] those graphs are slightly more concrete than in [3]. In [7] a program is translated to a constraint, whose solution is the set of *escaping* variables.

Here, we have defined the *escape property* $\mathcal{E}$ as a property of concrete states (Definition 11), and shown that it induces a non-trivial analysis, 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 9 and 13);
- That set is useful, sometimes, to restrict the possible targets of a virtual call *i.e., escape information contains class information* (Example 13).

In Figure 6, we compare the escape analyses presented in [2,3,7,11,14] with our own ($\mathcal{E}$). Here *context-sensitive* means that the analysis can provide different information in different program points; *compositional*, that the analysis of a compound command is defined in terms of the analysis of its components; *approximating the access to a field*, that the analysis can provide a better approximation of the creation points of the objects stored in a field than the set of all creation points; and *restricting the virtual calls*, that the analysis is able to discard some targets of a virtual call which can never be selected at run-time. These two last aspects contribute to the precision of the analysis.

The escape analysis induced by our domain $\mathcal{E}$ is not precise enough for practical use. However, the bottom line of Figure 6 shows that it is sometimes more precise than the other escape analyses for object-oriented languages developed up to now. Since $\mathcal{E}$ coincides with the *escape property* itself, either these other analyses are not a concretisation of this property, or their operations do not fully preserve it. We claim therefore that $\mathcal{E}$ makes a good starting point for a formal foundation of escape analysis. More precise domains are needed, but they should be *refinements* of $\mathcal{E}$. For an example of such refinements, see [8].

References

1. 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.
2. 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.
3. 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.
4. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998.
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 Refinement of the Escape Property. In A. Cortesi, editor, *Proc. of the VMCAI'02 workshop on Abstract Interpretation and Model Checking*, Venice, Italy, January 2002. To appear in Lecture Notes in Computer Science. Available at `http://www.sci.univr.it/~spoto/papers.html`.
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. Y. G. Park and B. Goldberg. Escape Analysis on Lists. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, volume 27(7) of *SIGPLAN Notices*, pages 116–127, San Francisco, California, USA, June 1992.
11. 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.
12. F. Spoto. The LOOP Analyser. `http://www.sci.univr.it/~spoto/loop/`.
13. 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.
14. 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.

A Proofs (not meant for publication)

Proof *of Lemma 1 at page IX*

∘ Since $\phi \cdot \mu \in \Sigma_\tau$, from Definition 8 we have $\phi \cdot \mu : \tau$. From Definition 7 we know that $o.\phi$ is weakly $F(o.\pi)$-correct *w.r.t.* μ. We conclude that $o.\phi \cdot \mu : F(o.\pi)$. Since $\mathtt{this} \notin \mathsf{dom}(F(o.\pi))$ (Definition 2) we conclude that $o.\phi \cdot \mu \in \Sigma_{F(o.\pi)}$. •

Proof *of Proposition 1 at page X*

∘ Since $\delta_\tau = \delta_\tau^{\#\Pi}$, it is enough to prove the result for δ_τ^i only. By Definition 10, the maps δ_τ^i for $i \in \mathbb{N}$ are reductive and monotonic. We prove idempotency by induction over $i \in \mathbb{N}$. Let $S \subseteq \Pi$. We have $\delta_\tau^0 \delta_\tau^0(S) = \delta_\tau^0(\varnothing) = \varnothing = \delta_\tau^0(S)$. Assume that the result holds for a given $i \in \mathbb{N}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and there is no $\pi \in S$ such that $k(\pi) \leq \tau(\mathtt{this})$, then $\delta_\tau^i \delta_\tau^i(S) = \delta_\tau^i(\varnothing) = \varnothing = \delta_\tau^i(S)$. Otherwise, by reductivity we have $\delta_\tau^{i+1}\delta_\tau^{i+1}(S) \subseteq \delta_\tau^{i+1}(S)$. We prove that also the converse inclusion holds. We have

$$\delta_\tau^{i+1}\delta_\tau^{i+1}(S) = \cup \left\{ \{\pi\} \cup \delta_{F(\pi)}^i \delta_\tau^{i+1}(S) \,\middle|\, \begin{array}{l} \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K} \\ \pi \in \delta_\tau^{i+1}(S),\ k(\pi) \leq \kappa \end{array} \right\} . \tag{4}$$

Let $\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K}$ and $\pi \in \Pi$ be such that $k(\pi) \leq \kappa$. If $\pi \in \delta_\tau^{i+1}(S)$ then, by reductivity, we have $\pi \in S$. Conversely, if $\pi \in S$ then, by Definition 10, $\pi \in \delta_\tau^{i+1}(S)$. We conclude that (4) is equal to

$$\begin{aligned} &\cup\{\{\pi\} \cup \delta_{F(\pi)}^i \delta_\tau^{i+1}(S) \mid \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in S,\ k(\pi) \leq \kappa\} \\ (*) \supseteq\ &\cup\{\{\pi\} \cup \delta_{F(\pi)}^i \delta_{F(\pi)}^i(S) \mid \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in S,\ k(\pi) \leq \kappa\} \\ (**) =\ &\cup\{\{\pi\} \cup \delta_{F(\pi)}^i(S) \mid \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ \pi \in S,\ k(\pi) \leq \kappa\} = \delta_\tau^{i+1}(S) , \end{aligned}$$

where point $*$ follows by Definition 10 and monotonicity and point $**$ follows by inductive hypothesis. •

Lemma A. *Let $\tau \in TypEnv$, $\sigma \in \Sigma_\tau$ and $i \in \mathbb{N}$. We have $\alpha_\tau^i(\sigma) = \delta_\tau^i \alpha_\tau^i(\sigma)$.*

Proof. By reductivity (Proposition 1) we have $\alpha_\tau^i(\sigma) \supseteq \delta_\tau^i \alpha_\tau^i(\sigma)$. Then it is enough to prove the converse inclusion. Let $\sigma = \phi \cdot \mu$. 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 $S = \{\mu\phi(v) \mid v \in \mathsf{dom}(\phi) \text{ and } \phi(v) \in Loc\}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and there is no $\pi \in \alpha_\tau^{i+1}(\sigma)$ such that $k(\pi) \leq \tau(\mathtt{this})$, then $\alpha_\tau^{i+1}(\sigma) = \varnothing = \delta_\tau^{i+1}\alpha_\tau^{i+1}(\sigma)$, because $\mathtt{this}$ cannot be bound to *nil* (Definition 8). Otherwise, by inductive hypothesis we have

$$\begin{aligned} \alpha_\tau^{i+1}(\sigma) &= \cup\{\{o.\pi\} \cup \alpha_{\tau'}^i(o.\phi \cdot \mu) \mid o \in S\} \\ &= \cup\{\{o.\pi\} \cup \delta_{\tau'}^i \alpha_{\tau'}^i(o.\phi \cdot \mu) \mid o \in S\} . \end{aligned} \tag{5}$$

By Definition 9, we have $\alpha^i_{\tau'}(o.\phi \cdot \mu) \subseteq \alpha^{i+1}_{\tau}(\sigma)$ and, by Proposition 1, (5) is contained in

$$\cup\{\{o.\pi\} \cup \delta^i_{\tau'}\alpha^{i+1}_{\tau}(\sigma) \mid o \in S\} \ . \tag{6}$$

Note that given $o \in S$ we can always find $\kappa \in \mathsf{rng}(\tau) \cap \mathcal{K}$ such that $k(o.\pi) \leq \kappa$. Indeed, for the definition of S there exists $v \in \mathsf{dom}(\phi) = \mathsf{dom}(\tau)$ such that $\phi(v) \in \mathit{Loc}$ and $o = \mu\phi(v)$. By Definition 5 we have $\tau(v) \in \mathcal{K}$. By Definition 7 we have $k(o.\pi) = k((\mu\phi(v)).\pi) \leq \tau(v)$. Then it is enough to choose $\kappa = \tau(v)$ and (6) is equal to

$$\begin{aligned}
&\cup\,\{\{o.\pi\} \cup \delta^i_{\tau'}\alpha^{i+1}_{\tau}(\sigma) \mid o \in S,\ \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ k(o.\pi) \leq \kappa\}\\
\text{(Definition 9)} \subseteq\ &\cup\,\{\{\pi\} \cup \delta^i_{\tau'}\alpha^{i+1}_{\tau}(\sigma) \mid \pi \in \alpha^{i+1}_{\tau}(\sigma),\ \kappa \in \mathsf{rng}(\tau) \cap \mathcal{K},\ k(\pi) \leq \kappa\}\\
\text{(Definition 10)} =\ &\delta^{i+1}_{\tau}\alpha^{i+1}_{\tau}(\sigma) \ .
\end{aligned}$$

•

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

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

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

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

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

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

Lemma B. *Let $\tau \in \mathit{TypEnv}$, $S_1, S_2 \subseteq \Pi$, $\phi \in \overline{\phi}_\tau(S_1)$ and $\mu \in \overline{\mu}(S_2)$. We have*

i) $\phi \cdot \mu : \tau$;
ii) $\phi \cdot \mu \in \Sigma_\tau$ iff `this` $\notin \mathsf{dom}(\tau)$ or there exists $\pi \in S_1$ s.t. $k(\pi) \leq \tau(\mathtt{this})$;
iii) If $\phi \cdot \mu \in \Sigma_\tau$ then $\alpha_\tau(\phi \cdot \mu) \subseteq S_1 \cup S_2$.

Proof.

i) Condition 1 of Definition 7 is satisfied because we have $\mathsf{rng}(\phi) \cap \mathit{Loc} \subseteq \{l_1, \ldots, l_n\} = \mathsf{dom}(\mu)$. Moreover, if $v \in \mathsf{dom}(\phi)$ and $\phi(v) \in \mathit{Loc}$ then $\phi(v) \in L_\tau(S_1, v)$. Thus there exists $1 \leq i \leq n$ such that $\phi(v) = l_i$, $(\mu\phi(v)).\pi = \pi_i$ and $k((\mu\phi(v)).\pi) = k(\pi_i) \leq \tau(v)$. Condition 2 of Definition 7 holds because if $o \in \mathsf{rng}(\mu)$ then $o.\phi = \phi_i$ for some $1 \leq i \leq n$. Since $\phi_i \in \overline{\phi}_{F(\pi_i)}(S)$, reasoning as above we conclude that ϕ_i is $F(\pi_i)$-correct *w.r.t.* μ. Then $\phi \cdot \mu : \tau$.

ii) By point i, we know that $\phi \cdot \mu : \tau$. From Definition 8, we have $\phi \cdot \mu \in \Sigma_\tau$ if and only if $\mathtt{this} \notin \mathsf{dom}(\tau)$ or $\phi(\mathtt{this}) \neq nil$. By Definition A, the latter case holds if and only if $L_\tau(S_1, \mathtt{this}) \neq \varnothing$ *i.e.*, if and only if there exists $\pi \in S_1$ such that $k(\pi) \leq \tau(\mathtt{this})$.

iii) Since $\phi \cdot \mu \in \Sigma_\tau$, the α_τ map is defined (Definition 9). Let

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

Since $\phi \in \overline{\phi}_\tau(S_1)$ and $o.\phi \in \overline{\phi}_{F(o.\pi)}(S_2)$ for every $o \in \mathsf{rng}(\mu)$, by Definition A we have

$$\{\mu(l).\pi \mid l \in L\} \subseteq S_1 \cup S_2\ .$$

By Definition 9 we conclude that

$$\alpha_\tau(\phi \cdot \mu) \subseteq \{\mu(l).\pi \mid l \in L\} \subseteq S_1 \cup S_2\ .$$

•

The following lemma shows that we know the abstraction of sets of states constructed from the frames and memories of Definition A.

Lemma C. *Let $\tau \in TypEnv$, $S_1, S_2 \subseteq \Pi$, $j \in \mathbb{N}$ and*

$$A^j = \alpha_\tau^{j+1}(\{\phi \cdot \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(S_1) \text{ and } \mu \in \overline{\mu}(S_2)\})\ .$$

Then

$$A^j = \begin{cases} \varnothing & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and there is no } \pi \in S_1 \text{ s.t. } k(\pi) \leq \tau(\mathtt{this}) \\ \cup\left\{ \{\pi\} \cup \delta^j_{F(\pi)}(S_2) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ \pi \in S_1,\ k(\pi) \leq \tau(v) \end{array} \right\} & \text{otherwise.} \end{cases}$$

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

$$A^0 = \begin{cases} \varnothing & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and there is no } \pi \in S_1 \text{ s.t. } k(\pi) \leq \tau(\mathtt{this}) \\ \left\{ o.\pi \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_\tau(S_1),\ \mu \in \overline{\mu}(S_2),\ v \in \mathsf{dom}(\phi) \\ \phi(v) \in Loc,\ o = \mu\phi(v) \end{array} \right\} & \text{otherwise.} \end{cases}$$

By Definition A, the latter case is equal to

$$\left\{ \pi_i \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ 1 \leq i \leq n,\ \pi_i \in S_1 \\ k(\pi_i) \leq \tau(v) \end{array} \right\} = \cup\left\{ \{\pi\} \cup \delta^0_{F(\pi)}(S_2) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau) \\ \tau(v) \in \mathcal{K},\ \pi \in S_1 \\ k(\pi) \leq \tau(v) \end{array} \right\}.$$

Assume now that the result holds for a given $j \in \mathbb{N}$. If $\mathtt{this} \in \mathsf{dom}(\tau)$ and there is no $\pi \in S_1$ such that $k(\pi) \leq \tau(\mathtt{this})$, by Lemma B.ii we have $A^{j+1} = \varnothing$. Otherwise

$$A^{j+1} = \cup\left\{ \{o.\pi\} \cup \alpha^{j+1}_{F(o.\pi)}(o.\phi \cdot \mu) \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_\tau(S_1),\ \mu \in \overline{\mu}(S_2),\ v \in \mathsf{dom}(\phi) \\ \phi(v) \in Loc,\ o = \mu\phi(v) \end{array} \right\}. \quad (7)$$

As for the base case, we know that $o.\pi$ ranges over $\{\pi \in S_1 \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K},\ k(\pi) \leq \tau(v)\}$. Since $o.\phi \in \overline{\phi}_{F(o.\pi)}(S_2)$ is arbitrary (Definition A), by inductive hypothesis (7) becomes

$$\cup \left\{ \{\pi\} \cup \alpha^{j+1}_{F(\pi)} \left(\left\{ \phi \cdot \mu \,\middle|\, \begin{array}{l} \phi \in \overline{\phi}_{F(\pi)}(S_2) \\ \mu \in \overline{\mu}(S_2) \end{array} \right\} \right) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K} \\ \pi \in S_1,\ k(\pi) \leq \tau(v) \end{array} \right\}$$
$$= \cup \left\{ \{\pi\} \cup \delta^{j+1}_{F(\pi)}(S_2) \mid v \in \mathsf{dom}(\tau),\ \tau(v) \in \mathcal{K},\ \pi \in S_1,\ k(\pi) \leq \tau(v) \right\} .$$

•

Corollary A. *Let $\tau \in \mathit{TypEnv}$ and $S_1, S_2 \subseteq \Pi$. Let*

$$A_\tau(S_1, S_2) = \alpha_\tau(\{\phi \cdot \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(S_1) \text{ and } \mu \in \overline{\mu}(S_2)\}) .$$

We have

i) $A_\tau(S_1, S_2) = \begin{cases} \varnothing & \text{if } \mathtt{this} \in \mathsf{dom}(\tau) \text{ and no } \pi \in S_1 \text{ is s.t. } k(\pi) \leq \tau(\mathtt{this}) \\ \cup \left\{ \{\pi\} \cup \delta_{F(\pi)}(S_2) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\phi),\ \tau(v) \in \mathcal{K} \\ \pi \in S_1,\ k(\pi) \leq \tau(v) \end{array} \right\} & \text{otherwise,} \end{cases}$

ii) $A_\tau(S_1, S_1) = \delta_\tau(S_1)$.

Proof. Point i follows by Lemma C since j is arbitrary. Point ii follows from point i and Definition 10. •

Corollary B. *Let $\kappa \in \mathcal{K}$, $\tau = [\mathit{res} \mapsto \kappa]$, p be a predicate over Π and $S \subseteq \Pi$ be such that there exists $\pi \in S$ such that $k(\pi) \leq \tau(\mathit{res})$ and $p(\pi)$ holds. We have*

$$\alpha_\tau(\{\phi \cdot \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(S),\ \mu \in \overline{\mu}(S),\ p(\mu\phi(\mathit{res}).\pi)\})$$
$$= \cup\{\{\pi\} \cup \delta_{F(\pi)}(S) \mid \pi \in S,\ k(\pi) \leq \tau(\mathit{res}),\ p(\pi)\} .$$

Proof. Let $j \in \mathbb{N}$. By the hypothesis on S we have

$$\alpha^{j+1}_\tau(\{\phi \cdot \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(S),\ \mu \in \overline{\mu}(S),\ p(\mu\phi(\mathit{res}).\pi)\})$$
$$= \cup\{\{o.\pi\} \cup \alpha^j_{F(o.\pi)}(o.\phi \cdot \mu) \mid \phi \in \overline{\phi}_\tau(S),\ \mu \in \overline{\mu}(S),\ o = \mu\phi(\mathit{res}),\ p(o.\pi)\}$$
$$= \cup\{\{\pi\} \cup \alpha^j_{F(\pi)}(\phi' \cdot \mu) \mid \pi \in S,\ k(\pi) \leq \tau(\mathit{res}),\ p(\pi),\ \phi' \in \overline{\phi}_{F(\pi)}(S),\ \mu \in \overline{\mu}(S)\}.$$

Since j is arbitrary we have

$$\alpha_\tau(\{\phi \cdot \mu \in \Sigma_\tau \mid \phi \in \overline{\phi}_\tau(S),\ \mu \in \overline{\mu}(S),\ p(\mu\phi(\mathit{res}).\pi)\})$$
$$= \cup \left\{ \{\pi\} \cup \alpha_{F(\pi)} \left(\left\{ \phi' \cdot \mu \,\middle|\, \begin{array}{l} \phi' \in \overline{\phi}_{F(\pi)}(S) \\ \mu \in \overline{\mu}(S) \end{array} \right\} \right) \,\middle|\, \begin{array}{l} \pi \in S,\ p(\pi) \\ k(\pi) \leq \tau(\mathit{res}) \end{array} \right\} ,$$

and the thesis follows by Corollary A.ii. •

Proof *of Lemma 2 at page XI*

∘ We first prove that $\mathsf{fp}(\delta_\tau) = \mathsf{rng}(\alpha_\tau)$.

Let $X \subseteq \Sigma_\tau$ and $i \in \mathbb{N}$. By Lemma A and monotonicity (Proposition 1) 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 opposite inclusion holds because δ_τ is reductive (Proposition 1). 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 $S \in \mathsf{fp}(\delta_\tau)$ and

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

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

Since δ_τ is reductive (Proposition 1), 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 8). Then $\alpha_\tau(X) = \varnothing$ if and only if $X = \varnothing$. •

Corollary C. *Let $\tau \in TypEnv$, $X \subseteq \Sigma_\tau$ and $S \subseteq \Pi$. Then $\alpha_\tau(X) \subseteq \delta_\tau(S)$ if and only if $\alpha_\tau(X) \subseteq S$.*

Proof. Assume that $\alpha_\tau(X) \subseteq \delta_\tau(S)$. By reductivity (Proposition 1) we have $\alpha_\tau(X) \subseteq S$. Conversely, assume that $\alpha_\tau(X) \subseteq S$. By Lemma 2 and monotonicity (Proposition 1) we have $\alpha_\tau(X) = \delta_\tau\alpha_\tau(X) \subseteq \delta_\tau(S)$. •

The following lemma will be used in the proof of Proposition 3. It states that integer values, *nil* and the name of the variables are not relevant to the definition of α (Definition 9).

Lemma D. *Let $\tau', \tau'' \in TypEnv$, $\phi' \cdot \mu \in \Sigma_{\tau'}$ and $\phi'' \cdot \mu \in \Sigma_{\tau''}$ such that $\mathsf{rng}(\phi') \cap Loc = \mathsf{rng}(\phi'') \cap Loc$. We have $\alpha_{\tau'}(\phi' \cdot \mu) = \alpha_{\tau''}(\phi'' \cdot \mu)$.*

Proof. From Definition 9. •

The following lemma relates the abstraction of a set of states with the abstraction of the same states whose frame has been restricted.

Lemma E. *Let $\tau \in TypEnv$ and $vs \subseteq \mathsf{dom}(\tau)$. We have*

$$\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \cdot \mu \mid \phi \cdot \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \cdot \mu) \subseteq S\}) = \delta_{\tau|_{-vs}}(S) \ .$$

Proof. We have

$$\begin{aligned} &\alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \cdot \mu \mid \phi \cdot \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \cdot \mu) \subseteq S\}) \\ &= \alpha_{\tau|_{-vs}}(\{\phi|_{-vs} \cdot \mu \in \Sigma_{\tau|_{-vs}} \mid \phi \cdot \mu \in \Sigma_\tau \text{ and } \alpha_\tau(\phi \cdot \mu) \subseteq S\}) \ , \end{aligned} \tag{8}$$

since if $\phi \cdot \mu \in \Sigma_\tau$ then $\phi|_{-vs} \cdot \mu \in \Sigma_{\tau|_{-vs}}$. We have that if $\alpha_\tau(\phi \cdot \mu) \subseteq S$ then $\alpha_{\tau|_{-vs}}(\phi|_{-vs} \cdot \mu) \subseteq S$. Hence (8) is contained in S. By Corollary C, (8) is also contained in $\delta_{\tau|_{-vs}}(S)$. But also the converse inclusion holds, since in (8) we can restrict the choice of $\phi \cdot \mu \in \Sigma_\tau$, so that (8) contains

$$\alpha_{\tau|_{-vs}}\left(\left\{\phi|_{-vs} \cdot \mu \in \Sigma_{\tau|_{-vs}} \,\middle|\, \begin{array}{l}\phi \cdot \mu \in \Sigma_\tau,\ \alpha_\tau(\phi \cdot \mu) \subseteq S \\ \phi \in \overline{\phi}_\tau(S),\ \mu \in \overline{\mu}(S)\end{array}\right\}\right) . \tag{9}$$

By points ii and iii of Lemma B, (9) is equal to

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

•

Proof *of Proposition 3 at page XI*

∘ By the theory of abstract interpretation [5], given $\tau \in \mathit{TypEnv}$ and $S \in \mathcal{E}_\tau$, the concretisation map induced by the abstraction map of Definition 9 is

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

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

We consider every operation in Figure 4 and we compute the induced optimal abstract operation, which will always coincide with that reported in Figure 5.

Figure 7 reports the signatures and the constraints over the operations in Figure 4. They are needed to follow the notation of this proof. Note that all the operations in Figure 4 use states in Σ_τ with $\mathtt{this} \in \mathsf{dom}(\tau)$. By Lemma 2 we have $\gamma_\tau(\varnothing) = \varnothing$. Then the powerset extension of the operations in Figure 4 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 7). The operation $\cup$ is not the powerset extension of an operation in Figure 4. 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 4 are not $\varnothing$ (except for the second argument of return and for $\cup$).

In this proof, we will use the following properties.

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

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

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

P1 holds since $S = \delta_\tau(S)$ (Definition 11). Then by Definition 10 we conclude that there exists such a π. P2 is a consequence of P1. Indeed, if π is like in P1, then $\sigma = [\mathtt{this} \mapsto l] \cdot [l \mapsto \pi \cdot \mathsf{init}(F(\pi))]$ with $l \in \mathit{Loc}$ belongs to Σ_τ for the definition of π and is such that (Definition 9) $\alpha_\tau(\sigma) = \{\pi\} \subseteq S$. P3 holds in every Galois insertion (Proposition 2).

Operation or predicate	Constraint for applicability ($\mathtt{this} \in \mathsf{dom}(\tau)$ always)
$\mathsf{nop}_\tau : \Sigma_\tau \mapsto \Sigma_\tau$	
$\mathsf{get\_int}^i_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res\mapsto int]}$	$res \notin \mathsf{dom}(\tau),\ i \in \mathbb{Z}$
$\mathsf{get\_nil}^\kappa_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res\mapsto \kappa]}$	$res \notin \mathsf{dom}(\tau),\ \kappa \in \mathcal{K}$
$\mathsf{get\_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res\mapsto \tau(v)]}$	$res \notin \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau)$
$\mathsf{get\_field}^f_\tau : \Sigma_\tau \to \Sigma_{\tau[res\mapsto F(\tau(res))(f)]}$	$res \in \mathsf{dom}(\tau)$ $\tau(res) \in \mathcal{K}$ $f \in \mathsf{dom}(F(\tau(res)))$
$\mathsf{put\_var}^v_\tau : \Sigma_\tau \mapsto \Sigma_{\tau\vert_{-res}}$	$res \in \mathsf{dom}(\tau),\ v \in \mathsf{dom}(\tau)$ $v \neq res$ $\tau(res) \leq \tau(v)$
$\mathsf{put\_field}^f_{\tau,\tau'} : \Sigma_\tau \mapsto (\Sigma_{\tau'} \to \Sigma_{\tau\vert_{-res}})$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$ $f \in \mathsf{dom}(F(\tau(res)))$ $\tau' = \tau[res \mapsto t]$ with $t \leq F(\tau(res))(f)$
$=_\tau, +_\tau : \Sigma_\tau \mapsto (\Sigma_\tau \mapsto \Sigma_\tau)$	$res \in \mathsf{dom}(\tau),\ \tau(res) = int$
$\mathsf{is\_nil}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res\mapsto int]}$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$
$\mathsf{call}^{\nu,v_1,\ldots,v_n}_\tau : \Sigma_\tau \mapsto \Sigma_{P(\nu)\vert_{-\mathtt{out}}}$	$\{v_1,\ldots,v_n\} \subseteq \mathsf{dom}(\tau),\ \nu \in \mathcal{M}$ $res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$ $P(\nu) \setminus \{\mathtt{out}, \mathtt{this}\} = \{\iota_1,\ldots,\iota_n\}$ $\tau(res) \leq P(\nu)(\mathtt{this})$ $\tau(v_i) \leq P(\nu)(\iota_i)$ for $i = 1,\ldots,n$
$\mathsf{return}^\nu_\tau : \Sigma_\tau \mapsto (\Sigma_{P(\nu)\vert_{\mathtt{out}}} \to \Sigma_{\tau[res\mapsto P(\nu)(\mathtt{out})]})$	$res \in \mathsf{dom}(\tau),\ \nu \in \mathcal{M}$
$\mathsf{restrict}^{vs}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau\vert_{-vs}}$	$vs \subseteq \mathsf{dom}(\tau)$
$\mathsf{expand}^{v:t}_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[v\mapsto t]}$	$v \notin \mathsf{dom}(\tau),\ t \in \mathit{Type}$
$\mathsf{new}^\pi_\tau : \Sigma_\tau \mapsto \Sigma_{\tau[res\mapsto k(\pi)]}$	$res \notin \mathsf{dom}(\tau),\ \pi \in \Pi$
$\mathsf{lookup}^{m,\nu}_\tau : \wp(\Sigma_\tau)$	$res \in \mathsf{dom}(\tau),\ \tau(res) \in \mathcal{K}$ $\nu \in \mathcal{M}$ $m \in \mathsf{dom}(M(\tau(res)))$
$\mathsf{is\_true}_\tau, \mathsf{is\_false}_\tau : \wp(\Sigma_\tau)$	$res \in \mathsf{dom}(\tau),\ \tau(res) = int$

Figure 7. The signature of the operations over the states.

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

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

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

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

where point $*$ follows by Lemma D 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 $\mathsf{get\_nil}$ and of $\mathsf{get\_var}$.

<u>expand</u>

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

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

<u>restrict</u>

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

<u>is_nil</u>

$$\begin{aligned}
&\alpha_{\tau[res\mapsto int]}(\mathsf{is_nil}_\tau(\gamma_\tau(S)))\\
&= \alpha_{\tau[res\mapsto int]}(\mathsf{is_nil}_\tau(\{\sigma\in\Sigma_\tau \mid \alpha_\tau(\sigma)\subseteq S\}))\\
&= \alpha_{\tau[res\mapsto int]}(\{\phi[res\mapsto 1]\cdot\mu \mid \phi\cdot\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\cdot\mu)\subseteq S\})\\
(\text{Lemma D}) &= \alpha_{\tau|_{-res}}(\{\phi|_{-res}\cdot\mu \mid \phi\cdot\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\cdot\mu)\subseteq S\})\\
(\text{Lemma E}) &= \delta_{\tau|_{-res}}(S)\ .
\end{aligned}$$

<u>put_var</u>

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\mathsf{put_var}_\tau(\gamma_\tau(S)))\\
&= \alpha_{\tau|_{-res}}(\mathsf{put_var}_\tau(\{\sigma\in\Sigma_\tau \mid \alpha_\tau(\sigma)\subseteq S\}))\\
&= \alpha_{\tau|_{-res}}(\{\phi[v\mapsto\phi(res)]|_{-res}\cdot\mu \mid \phi\cdot\mu\in\Sigma_\tau \text{ and } \alpha_\tau(\phi\cdot\mu)\subseteq S\})\ . \qquad (10)
\end{aligned}$$

We have $\mathsf{rng}(\phi[v\mapsto\phi(res)]|_{-res}) = \mathsf{rng}(\phi|_{-v})$. Then, by Lemma D and Lemma E, (10) is equal to

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

<u>call</u>

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

where point $*$ follows by Lemma D and point $**$ follows by Lemma E.

<u>is_true, is_false</u>

$$\begin{aligned}
&\alpha_\tau(\mathsf{is_true}_\tau(\gamma_\tau(S)))\\
&= \alpha_\tau(\{\phi\cdot\mu \in \gamma_\tau(S) \mid \phi(res) \geq 0\})\\
(\text{Lemma D}) &= \alpha_\tau\gamma_\tau(S) = S\ .
\end{aligned}$$

The same proof holds for is_false.

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

$$\begin{aligned}
&\alpha_{\tau[res\mapsto\kappa]}(\mathsf{new}^\pi_\tau(\gamma_\tau(S)))\\
&= \alpha_{\tau[res\mapsto\kappa]}(\mathsf{new}^\pi_\tau(\{\sigma \in \Sigma_\tau \mid \alpha_\tau(\sigma) \subseteq S\}))\\
&= \alpha_{\tau[res\mapsto\kappa]}\left(\left\{\begin{array}{c} \phi[res\mapsto l]\cdot \\ \cdot\mu[l\mapsto \pi\cdot\mathsf{init}(F(\kappa))] \end{array}\middle| \begin{array}{l} \phi\cdot\mu \in \Sigma_\tau,\ \alpha_\tau(\phi\cdot\mu)\subseteq S \\ l \in Loc\setminus\mathsf{dom}(\mu)\end{array}\right\}\right)\\
&= \alpha_\tau(\{\phi\cdot\mu \in \Sigma_\tau \mid \alpha_\tau(\phi\cdot\mu)\subseteq S\})\cup && (11)\\
&\quad\cup\, \alpha_{[res\mapsto\kappa]}\left(\left\{\begin{array}{c} [res\mapsto l]\cdot \\ \cdot[l\mapsto \pi\cdot\mathsf{init}(F(\kappa))] \end{array}\middle| \begin{array}{l} \phi\cdot\mu \in \Sigma_\tau,\ \alpha_\tau(\phi\cdot\mu)\subseteq S \\ l \in Loc\setminus\mathsf{dom}(\mu)\end{array}\right\}\right). && (12)
\end{aligned}$$

We have that (11) is equal to S. By P2 and Definition 9, (12) is equal to $\{\pi\}$.

<u>=, +</u>

$$\begin{aligned}
&\alpha_\tau(=_\tau(\gamma_\tau(S_1))(\gamma_\tau(S_2)))\\
&= \alpha_\tau(\{=_\tau(\sigma_1)(\sigma_2) \mid \sigma_1 \in \gamma_\tau(S_1),\ \sigma_2 \in \gamma_\tau(S_2)\})\\
(\text{P2}) &= \alpha_\tau(\{\sigma_2 \mid \sigma_2 \in \gamma_\tau(S_2)\})\\
&= \alpha_\tau\gamma_\tau(S_2) = S_2\ .
\end{aligned}$$

A similar proof holds for $+$.

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

$$\begin{aligned}
&\alpha_{\tau'}(\mathsf{return}^\nu_\tau(\gamma_\tau(S_1))(\gamma_{\tau''}(S_2)))\\
&= \alpha_{\tau'}(\mathsf{return}^\nu_\tau(\{\sigma_1 \in \Sigma_\tau \mid \alpha_\tau(\sigma_1)\subseteq S_1\})(\{\sigma_2 \in \Sigma_{\tau''} \mid \alpha_{\tau''}(\sigma_2)\subseteq S_2\}))\\
&= \alpha_{\tau'}\left(\left\{\phi_1|_{-res}[res\mapsto\phi_2(\mathtt{out})]\cdot\mu_2 \middle| \underbrace{\begin{array}{c} \phi_1\cdot\mu_1\in\Sigma_\tau,\ \phi_2\cdot\mu_2\in\Sigma_{\tau''} \\ \alpha_\tau(\phi_1\cdot\mu_1)\subseteq S_1 \\ \alpha_{\tau''}(\phi_2\cdot\mu_2)\subseteq S_2,\ \mu_1 \lhd_L \mu_2 \end{array}}_{Cond}\right\}\right)\\
(*) &= \alpha_{\tau|_{-res}}(\{\phi_1|_{-res}\cdot\mu_2 \mid Cond\})\cup && (13)\\
&\quad\cup\,\alpha_{\tau''}(\{\phi_2\cdot\mu_2 \mid Cond\}) && (14)
\end{aligned}$$

where point $*$ follows by Lemma D. Since $\alpha_{\tau''}(\phi_2 \cdot \mu_2) \subseteq S_2$, an upper bound of (14) is S_2. But S_2 is also a lower bound of (14) since, by Lemma B.iii, a lower bound of (14) is

$$\alpha_{\tau''}\left(\left\{\phi_2 \cdot \mu_2 \,\middle|\, \begin{array}{l} \phi_1 \in \overline{\phi}_\tau(S_1),\ \mu_1 \in \overline{\mu}(S_1) \\ \phi_2 \in \overline{\phi}_{\tau''}(S_2),\ \mu_2 \in \overline{\mu}(S_2) \end{array}\right\}\right)$$

which by Corollary A.ii is equal to S_2. Note that the condition $\mu_1 \lhd_L \mu_2$ is satisfied by definition A.

Equation (13), instead, is

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

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

$$\begin{aligned}
&\cup\left\{\{o.\pi\} \cup \alpha_{F(o.\pi)}(o.\phi \cdot \mu_2) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\phi_1|_{-res}),\ \phi_1|_{-res}(v) \in Loc \\ o = \mu_1\phi_1|_{-res}(v),\ Cond \end{array}\right\} \\
(*) \subseteq &\cup\left\{\{o.\pi\} \cup \delta_{F(o.\pi)}(\Pi) \,\middle|\, \begin{array}{l} v \in \mathsf{dom}(\phi_1|_{-res}),\ \phi_1|_{-res}(v) \in Loc \\ o = \mu_1\phi_1|_{-res}(v),\ Cond \end{array}\right\} \\
(**) \subseteq &\cup\{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K},\ \pi \in S_1,\ k(\pi) \leq \kappa,\ Cond\} \\
\subseteq &\cup\{\{\pi\} \cup \delta_{F(\pi)}(\Pi) \mid \kappa \in \mathsf{rng}(\tau|_{-res}) \cap \mathcal{K},\ \pi \in S_1,\ k(\pi) \leq \kappa\}\ , \qquad (15)
\end{aligned}$$

where point $*$ follows by Lemma A and point $**$ holds since $Cond$ requires that $\alpha_\tau(\phi_1 \cdot \mu_1) \subseteq S_1$. But (15) is also a lower bound of (13), since (13) contains

$$\begin{aligned}
&\alpha_{\tau|_{-res}}\left(\left\{\phi_1|_{-res} \cdot \mu_2 \in \Sigma_{\tau|_{-res}} \,\middle|\, \begin{array}{l} \phi_1 \in \overline{\phi}_\tau(S_1),\ \mu_1 \in \overline{\mu}(S_1), \\ \phi_2 = \mathsf{init}(P(\nu)|_{\mathsf{out}}),\ \mu_2 \in \overline{\mu}(\Pi) \end{array}\right\}\right) \\
&\qquad = \alpha_{\tau|_{-res}}(\{\phi \cdot \mu \in \Sigma_{\tau|_{-res}} \mid \phi \in \overline{\phi}_{\tau|_{-res}}(S_1), \mu \in \overline{\mu}(\Pi)\})\ ,
\end{aligned}$$

which by Corollary A.i is equal to (15).

<u>get_field</u>
Let $\tau' = \tau[res \mapsto F(\tau(res))(f)]$ and $\tau'' = [res \mapsto F(\tau(res))(f)]$. We have

$$\begin{aligned}
&\alpha_{\tau'}(\mathsf{get_field}^f_\tau(\gamma_\tau(S))) \\
&= \alpha_{\tau'}(\mathsf{get_field}^f_\tau(\{\phi \cdot \mu \in \Sigma_\tau \mid \alpha_\tau(\phi \cdot \mu) \subseteq S\})) \\
&= \alpha_{\tau'}\left(\left\{\phi|_{-res}[res \mapsto (\mu\phi(res)).\phi(f)] \cdot \mu \,\middle|\, \begin{array}{l} \phi \cdot \mu \in \Sigma_\tau,\ \phi(res) \neq nil, \\ \alpha_\tau(\phi \cdot \mu) \subseteq S \end{array}\right\}\right) \\
&= \alpha_{\tau|_{-res}}(\{\phi|_{-res} \cdot \mu \mid \phi \cdot \mu \in \Sigma_\tau,\ \phi(res) \neq nil,\ \alpha_\tau(\phi \cdot \mu) \subseteq S\}) \cup \\
&\quad \cup\, \alpha_{\tau''}\left(\left\{[res \mapsto (\mu\phi(res)).\phi(f)] \cdot \mu \,\middle|\, \begin{array}{l} \phi \cdot \mu \in \Sigma_\tau,\ \phi(res) \neq nil, \\ \alpha_\tau(\phi \cdot \mu) \subseteq S \end{array}\right\}\right)\ . \qquad (16)
\end{aligned}$$

Equation (16) is equal to $\varnothing$ if $\{\pi \in S \mid k(\pi) \leq \tau(res)\} = \varnothing$, since in such a case the condition $\phi(res) \neq nil$ cannot be satisfied. Since $\alpha_\tau(\phi \cdot \mu) \subseteq S$, an upper bound of (16) is S. By Corollary C, also $\delta_{\tau'}(S)$ is an upper bound of (16). But it is also a lower bound of (16), since, from the hypothesis on S and from points ii and iii of Lemma B, (16) contains

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

where point $*$ follows by Definition A and point $**$ follows by Corollary A.ii.

<u>lookup</u>

$$
\begin{aligned}
&\alpha_\tau(\mathsf{lookup}_\tau^{m,\nu}(\gamma_\tau(S)))\\
&=\alpha_\tau(\mathsf{lookup}_\tau^{m,\nu}(\{\phi\cdot\mu\in\Sigma_\tau\mid\alpha_\tau(\phi\cdot\mu)\subseteq S\}))\\
&=\alpha_\tau(\{\phi\cdot\mu\in\Sigma_\tau\mid\underbrace{\alpha_\tau(\phi\cdot\mu)\subseteq S,\ \phi(res)\neq nil,\ M((\mu\phi(res)).\pi)(m)=\nu}_{Cond}\}).
\end{aligned}
\tag{17}
$$

Equation (17) is equal to $\varnothing$ if there is no $\pi\in S$ such that $k(\pi)\le\tau(res)$ 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

$$\alpha_{\tau|_{-res}}(\{\phi|_{-res}\cdot\mu\mid\phi\cdot\mu\in\Sigma_\tau,\ Cond\})\cup \tag{18}$$

$$\cup\,\alpha_{\tau|_{res}}(\{\phi|_{res}\cdot\mu\mid\phi\cdot\mu\in\Sigma_\tau,\ Cond\})\ . \tag{19}$$

Since $Cond$ requires that $\alpha_\tau(\phi\cdot\mu)\subseteq S$, by Corollary C an upper bound of (18) is $\delta_{\tau|_{-res}}(S)$. But it is also a lower bound of (18), since a lower bound of (18) is

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

Point $*$ follows from the hypothesis on S. Point $**$ follows by Corollary A.ii.

Equation (19), instead, is contained in

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

which, by Corollary B, is

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

put_field

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\mathsf{put_field}_{\tau,\tau'}(\gamma_\tau(S_1))(\gamma_\tau(S_2)))\\
&=\alpha_{\tau|_{-res}}(\mathsf{put_field}_{\tau,\tau'}(\{\sigma_1\in\Sigma_\tau\mid\alpha_\tau(\sigma_1)\subseteq S_1\})(\{\sigma_2\in\Sigma_{\tau'}\mid\alpha_{\tau'}(\sigma_2)\subseteq S_2\}))\\
&=\alpha_{\tau|_{-res}}\left(\left\{\begin{array}{l|l}\phi_2|_{-res}\cdot\mu_2[l\mapsto\mu_2(l).\pi\cdot & \phi_1\cdot\mu_1\in\Sigma_\tau,\ \phi_2\cdot\mu_2\in\Sigma_{\tau'}\\ \cdot\mu_2(l).\phi[f\mapsto\phi_2(res)]] & \alpha_\tau(\phi_1\cdot\mu_1)\subseteq S_1,\ \alpha_{\tau'}(\phi_2\cdot\mu_2)\subseteq S_2\\ & (l=\phi_1(res))\neq nil,\ \mu_1\lhd_l\mu_2\end{array}\right\}\right)
\end{aligned}\tag{20}$$

which is $\varnothing$ if there is no $\pi\in S_1$ 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 $\alpha_{\tau'}(\phi_2\cdot\mu_2)\subseteq S_2$, we conclude that an upper bound of (20) is S_2. Then $\delta_{\tau|_{-res}}(S_2)$ is also an upper bound of (20) (Corollary C). We show that it is also a lower bound. Let $\pi_1\in S_1$ be such that $k(\pi_1)\leq\tau(\mathtt{this})$ (possible for P1) and $\pi_2\in S_1$ be such that $k(\pi_2)\leq\tau(res)$ (possible for the hypothesis on S_1). Let $o_1=\pi_1\cdot\mathsf{init}(F(\pi_1))$ and $o_2=\pi_2\cdot\mathsf{init}(F(\pi_2))$. We obtain the following lower bound of (20) by choosing special cases for ϕ_1, μ_1, ϕ_2 and μ_2.

$$\alpha_{\tau|_{-res}}\left(\left\{\begin{array}{l|l} & \phi_1=\mathsf{init}(\tau)[\mathtt{this}\mapsto l_1,res\mapsto l_2]\\ \phi_2|_{-res}\cdot\mu_2[l_2\mapsto\mu_2(l_2).\pi\cdot & \phi_2\in\overline{\phi}_{\tau'}(S_2),\ \mu_2'\in\overline{\mu}(S_2)\\ \cdot\mu_2(l_2).\phi[f\mapsto\phi_2(res)]] & \phi_2\cdot\mu_2'\in\Sigma_{\tau'}\\ & \mu_1=\mu_2=\mu_2'[l_1\mapsto o_1,l_2\mapsto o_2]\\ & l_1,l_2\in Loc\setminus\mathsf{dom}(\mu_2'),\ l_1\neq l_2\end{array}\right\}\right).\tag{21}$$

Since l_2 is not used in ϕ_2 nor in μ_2', (21) becomes

$$\begin{aligned}
&\alpha_{\tau|_{-res}}(\{\phi_2|_{-res}\cdot\mu_2\in\Sigma_{\tau|_{-res}}\mid\phi_2\in\overline{\phi}_{\tau'}(S_2),\ \mu_2\in\overline{\mu}(S_2)\})\\
\text{(Definition A)}&=\alpha_{\tau|_{-res}}(\{\phi\cdot\mu\in\Sigma_{\tau|_{-res}}\mid\phi\in\overline{\phi}_{\tau|_{-res}}(S_2),\ \mu\in\overline{\mu}(S_2)\})\\
\text{(Lemma C)}&=\delta_{\tau|_{-res}}(S_2)\ .
\end{aligned}$$

$\cup$

By additivity (Proposition 2), the best approximation of $\cup$ in $\wp(\Sigma_\tau)$ is $\cup$ in $\wp(\Pi)$. •

Proof *of Proposition 4 at page XII*

∘ The result, from given inputs, of an operation in Figure 5 can be computed in time $O(p^2t)$. For instance, consider the lookup operation, which is the more complex one. The computation of S' and its test for emptiness can be computed in $O(p)$. Then we compute the δ function $\#S'+1$ times *i.e.,* $O(p)$ times. Computing δ requires at most p iterations, and each iteration can be computed in $O(t)$ (Definition 10). Then the cost for computing lookup from a given input is $O(p^2t)$. Simpler reasonings apply to the other operations in Figure 5. Since we have $O(2^p)$ possible inputs, the cost for computing the denotation of an operation in Figure 5 is $O(tp^2 2^p) = O(t2^p)$.

The cost in time for joining two denotations d_1 and d_2 (*i.e.*, for computing the denotation of the composition of two operations in Figure 5) is $O(2^p)$, since for every input i of d_1 we search $d_1(i)$ among the inputs of d_2. This search costs $O(2^p)$ and is performed $O(2^p)$ times (similarly for binary operations).

The cost in time for computing the denotation of a sequence of m commands is then $O(mt2^p + m2^p) = O(mt2^p)$, since we compute the denotation of every single command and then we join them. Note that a sequence of m commands is translated in a sequence of $O(m)$ operations from Figure 5 [13].

Let l be the number of strongly connected components of the call graph of the program, f be one of these components, b_f the number of nodes of f and n_f the total number of commands in the nodes of f. Every iteration of the local fixpoint over f costs $O(n_f t2^p)$ as shown before. At every iteration, the denotation of at least one node of f must change, and that denotation can change at most $p2^p$ times (every denotation has at most 2^p outputs and each output can take at most p different values). Then the local fixpoint is reached after at most $O(b_f p2^p)$ iterations. The total cost for its computation is then $O(b_f p2^p n_f t2^p) = O(b_f n_f t2^p)$.

The cost for computing the denotation (analysis) of the whole program is

$$\sum_{f=1}^{l} O(b_f n_f t2^p) = O\left(t2^p \cdot \sum_{f=1}^{l}(b_f n_f)\right) . \tag{22}$$

Since $\sum_{f=1}^{l}(b_f n_f) \leq \sum_{f=1}^{l}(b_f n) = n \cdot \sum_{f=1}^{l} b_f = bn$, (22) is $O(bnt2^p)$. •