Dominance Analysis via Ownership Types and Abstract Interpretation

Ilya Sergey[1], Jan Midtgaard[2], and Dave Clarke[1]

[1] IBBT-DistriNet, Department of Computer Science,
Katholieke Universiteit Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be
[2] Department of Computer Science, Aarhus University, Denmark
jmi@cs.au.dk

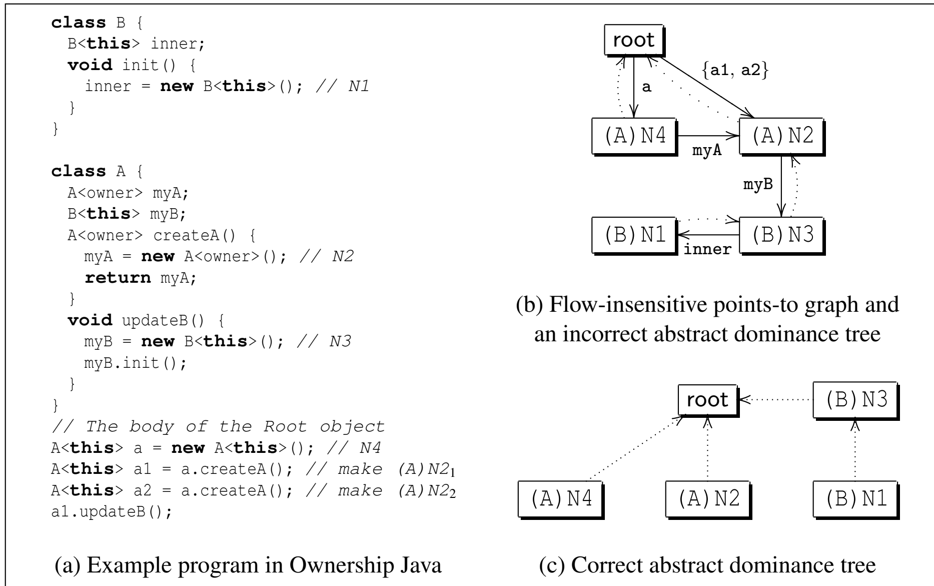
Abstract. Ownership types provide a declarative way to statically structure the topology of the heap and control aliasing in object-oriented programs. However, the relation between systematically derived static program analyses by abstract interpretation and semantic properties enforced by ownership types has not yet been investigated. In this work we build a framework to statically compute an abstract object dominance tree, based on the information provided by ownership types in the context of the *owners-as-dominators* policy. We develop a series of concrete and abstract domains that enable us to abstract a tree-like structure in a way that respects the *ancestor relation* between particular nodes of the tree and prove them to be Galois connections. We then plug the developed domains into a traditional abstract interpretation-based points-to analysis. The resulting abstract semantics is derived systematically from the concrete transition relation instrumented for tree computation and proven to be sound. The presented framework is *tunable* concerning polyvariance: the resulting abstract tree depends on the underlying context picking strategy.

Keywords: ownership types, abstract interpretation, points-to analysis, Galois connections, dominance, uniqueness, object-orientation

1 Introduction

Ownership types have been introduced into object-oriented languages in order to allow modular reasoning about aliasing and structure of the heap [4, 2]. Variants of ownership types allow a program to enjoy computational properties such as data race-freedom, disjointness of effects, various confinement properties and effective memory management. However, the modularity has a downside: the reasoning about points-to information, aliasing and heap structure via ownership types can be done only *locally*, i.e., in contexts where types are well-formed, e.g., methods and classes. Thus, given a program, consisting of a set of classes, it is hard to extract information about the relation of objects in different classes with respect to the type-enforced policy.

This sort of *global* information about a program is normally revealed by a series of static program analyses, such as control-flow or points-to analyses [15, 23, 24]. However, the existing analyses are not tailored to take the ownership information into account, which leads to imprecise and unsound results considering the property of interest.



(a) Example program in Ownership Java

(b) Flow-insensitive points-to graph and an incorrect abstract dominance tree

(c) Correct abstract dominance tree

Fig. 1: An abstract object graph and corresponding dominance trees (dotted)

In this work we explore the marriage of ownership types and a family of semantics-based points-to analyses via abstract interpretation [7, 8]. We focus on the *owners-as-dominators* property [3] ensured by the type annotations, and build an analysis for static detection of *whole-program* dominance patterns in object-oriented programs.

In order to do so, we develop a series of concrete and abstract domains that enable us to infer a tree-like structure on top of the *object graph* of the program in a way that respects the *ancestor relation* between particular nodes. We prove these domains to be Galois connections and then plug them into a traditional abstract interpretation-based points-to analysis. The resulting abstract semantics for the analysis is derived systematically from the concrete transition relation instrumented for tree computation and proven to be sound. The semantics is parametrized by context-sensitivity à la Shivers's k-CFA [23], in which a parameter can tune the precision of the computed abstraction.

A motivating example: abstracting dominance trees Figure 1 provides a motivating example for this work. The program Fig. 1(a) contains two classes: `A` and `B`. Whenever an instance of `A` or `B` is allocated, another object is assigned to it as its *owner*. This might be an object, referred to by some immutable local variable, a this-reference, an owner of the allocator object, which is denoted by `owner`, or root, which stands for the unique top object, whose method `main` is invoked by an interpreter. In this settings, the owners-as-dominators invariant (OAD) [4] can be informally stated as follows: in the object graph of the program rooted with root, there are no paths from root along field-references to an object o_1 from another object o_2 that bypass the owner of o_1. In other words, the owner of any object is its dominator in the object graph.

The OAD invariant is considered to be useful for eliminating synchronization locks in concurrent programs and memory management in real-time applications. This is a

good motivation for us to design an analysis that helps to answer the following question: "Whether *all* object instances allocated at the site N_1 in the context c_1 are dominated by *all* instances, allocated at N_2 in the context c_2?"

One can try to use traditional points-to analysis techniques to solve this problem. For instance, building an abstract points-to graph using monovariant *may*-points-to analysis for the considered program gives rise to Fig. 1(b), where all object instances are associated with their allocation sites. Solid arrows denote field references in the abstract object graph. The dominance tree for the graph (denoted by dotted arrows) can be computed by traditional means [16]. However, the resulting dominator tree is *incorrect* with respect to the concrete object graph, since the abstract object `(A)N2` is an ancestor (i.e., dominator) of `(B)N3`, whereas in the concrete execution it is not the case: only one of two objects, allocated at `N2`, namely `(A)N2`$_1$, dominates `(B)N3`. In the absence of some auxiliary context information, which could increase precision of the analysis, the correct dominance tree for this case is present at Fig. 1(c). Note, that the edge `(B)N1` $\rightarrow$ `(B)N3` is *correct*, since in the concrete program execution *all* objects allocated at `N3` dominate *all* objects, allocated at `N1`. Employing *must*-analysis also does not provide a perfect solution, since it can give rise to non-connected points-to graphs.

The cause of the observed problem is the fact that the traditional domain for the collecting semantics is not tailored to maintain dominance information in a way that can be abstracted. Since information about owners of abstract objects can be lost because of "merging" while abstracting program states, we consider *some* known-to-be-correct dominator of an abstract object as an abstraction of its *immediate* dominator, i.e., its owner. In the following sections we put this observation into the core of the *dominance* analysis. In summary, this paper makes the following contributions:

- We describe and formalize two domains for maintaining information about ancestors in tree-like structures, which soundly approximate the ancestor relation, and we prove these domains to be Galois connections;
- We formalize the concrete collecting semantics for computations that maintain the tree of owners;
- We systematically build an abstract semantics for tunable dominance analysis by abstracting the concrete semantics and prove it to be a sound approximation. The developed analysis takes a program in a subset of Java with ownership annotations as an input and returns a whole-program abstract dominance tree as an output;
- Finally, we discuss other applications of the developed framework, e.g., for type inference and call graph analysis.

2 Abstractions for Trees and Uniqueness

In this section, we describe two families of domains: *tree domains* and *uniqueness domains*. The first family is targeted toward maintaining evolving tree-like structures and is inspired by *ownership functions*, introduced in Wren's master's thesis [26]. Because of the abstraction in points-to analyses, multiple concrete objects may correspond to a single abstract object, which complicates maintaining the ordering of nodes in an abstract tree. In order to tackle this problem, we describe the second family of domains,

which is aimed to track this fact similarly to the singleton abstraction typical for shape analyses [13].

2.1 Tree functions and tree domains

Definition 1 (Tree functions). *A triple* $\langle S, \mathsf{root}, \theta \rangle$ *defines a* tree function *on the set* S *iff* $\mathsf{root} \in S$, $\theta : S \to S$ *and*

1. $\theta(\mathsf{root}) = \mathsf{root}$
2. $\forall a \in S\ \exists k \geq 0 : \theta^k(a) = \mathsf{root}$

The following proposition states the correspondence between tree functions and traditional definition of a tree from graph theory.

Proposition 1. *A tree function* $\langle S, \mathsf{root}, \theta \rangle$ *defines a tree structure* τ *with the root* root *and nodes from* S*, such that for all* $a \in S$, $a \neq \mathsf{root} : \theta(a) = \mathsf{parent}(a)$.

Proof. Using the definition of a tree as an undirected graph, which is (a) *connected* and (b) *has no cycles* [6], we show by contradiction that

(a) τ is connected, otherwise root is not reachable from some nodes by iterating $\theta \Rightarrow$ a contradiction;
(b) τ is acyclic. Assuming the existence of a simple cycle in τ (otherwise just break the cycle into simple ones), there must be $a, a_1, a_2 \in S$, such that $\theta(a) = a_1$ and $\theta(a) = a_2$ and $a_1 \neq a_2$, which contradicts to the fact that θ is a *function* $\Rightarrow$ a contradiction. □

In the rest of the paper we will abuse notation, referring to some θ as a tree function, assuming a triple $\langle S, \mathsf{root}, \theta \rangle$, such that $S = \mathsf{dom}(\theta)$, $\mathsf{root} \in S$ and θ satisfies the conditions of Definition 1. Employing the correspondence from Proposition 1, we will use the following definition of θ-children:

$$\forall\ a \neq \mathsf{root},\ children_\theta(a) = \{a' \in \mathsf{dom}(\theta) \mid \theta(a') = a\}.$$

A tree function θ with $S = \mathsf{dom}(\theta)$ induces a partial order $\sqsubseteq_\theta$ on S, i.e., for all $a_1, a_2 \in S$

$$a_1 \sqsubseteq_\theta a_2 \triangleq \exists k \geq 0 : a_2 = \theta^k(a_1)$$

Given $a_1, a_2 \in S$ and a tree function θ, such that $S = \mathsf{dom}(\theta)$, we say that a_2 θ-*antecedes* a_1 iff $a_1 \sqsubseteq_\theta a_2$. We will be using the term "antecedence" when talking about elements of the domain of some tree function without specifying the function if it is clear from the context. The tree-order $\sqsubseteq_\theta$ induces the least upper bound $\sqcup_\theta$, which is defined naturally as a *least common* θ*-ancestor* of two elements: $a_1 \sqcup_\theta a_2 \triangleq a \in \mathsf{dom}(\theta)$, such that $a_1 \sqsubseteq_\theta a$ and $a_2 \sqsubseteq_\theta a$ and $\forall a' : (a_1 \sqsubseteq_\theta a' \ \wedge\ a_2 \sqsubseteq_\theta a') \Rightarrow a \sqsubseteq_\theta a'$.

Definition 2 (Tree domains). *Given a set* A*, such that* $\mathsf{root} \in A$*, then* $\langle \mathcal{T}(A), \sqsubseteq_\mathcal{T} \rangle$ *is a tree domain, where* $\mathcal{T}(A) = \{\theta \mid S \subseteq A$ *and* $\langle S, \mathsf{root}, \theta \rangle$ *defines a tree function*$\}$ *and the relation* $\sqsubseteq_\mathcal{T}$ *is defined by the following rule:*

$$\textsc{(Tree-Prec)}\quad \frac{\mathsf{dom}(\theta_1) \subseteq \mathsf{dom}(\theta_2) \qquad \forall a \in \mathsf{dom}(\theta_1)\ \exists k > 0 : \theta_1^k(a) = \theta_2(a)}{\theta_1 \sqsubseteq_\mathcal{T} \theta_2}$$

Proposition 2. *Given a set A, then $\sqsubseteq_{\mathcal{T}}$ is a partial order on $\mathcal{T}(A)$.*[3]

In the terms of Definition 2, define $\sqcup_{\mathcal{T}}$, $\top_{\mathcal{T}}$ and $\bot_{\mathcal{T}}$ as follows:

$$\top_{\mathcal{T}} = \lambda a.\ \mathsf{root} \qquad \bot_{\mathcal{T}} = \lambda a. \begin{cases} \mathsf{root} & \text{if } a = \mathsf{root} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$\theta_1 \sqcup_{\mathcal{T}} \theta_2 = \lambda a. \begin{cases} \theta_1(a) & \text{if } a \in \mathsf{dom}(\theta_1) \setminus \mathsf{dom}(\theta_2) \\ \theta_2(a) & \text{if } a \in \mathsf{dom}(\theta_2) \setminus \mathsf{dom}(\theta_1) \\ a' : a \sqsubseteq_{\theta_1} a' \wedge a \sqsubseteq_{\theta_2} a' \wedge & \\ \quad \forall k \geq 0 : \theta_1^k(a') = \theta_2^k(a') & \text{otherwise} \end{cases}$$

The last clause of the definition of $\sqcup_{\mathcal{T}}$ finds the least upper bound of two tree functions by "pulling" the nodes until the greatest common sub-branch in both trees, induced by θ_1 and θ_2 will be found. It is easy to show that $\langle \mathcal{T}(A), \sqsubseteq_{\mathcal{T}} \rangle$ is a semilattice. The following lemma relates the tree order and antecedence in tree functions.

Lemma 1 (Tree order and antecedence). *If $\theta_1 \sqsubseteq_{\mathcal{T}} \theta_2$ iff for all $a_1, a_2 \in \mathsf{dom}(\theta_1)$ $a_1 \sqsubseteq_{\theta_2} a_2 \Rightarrow a_1 \sqsubseteq_{\theta_1} a_2$.*

Proof. ($\Rightarrow$). By induction on $k \geq 0$ such that $a_2 = \theta_2^k(a_1)$. The case $k = 0$ is trivial. The case $k = 1$ matches the rule (TREE-PREC), and for $k > 1$ one should apply the induction hypothesis to $a' = \theta_2(a_1)$. Since $a_1 \sqsubseteq_{\theta_1} a'$ and $a' \sqsubseteq_{\theta_2} a_2$, conclude $a' \sqsubseteq_{\theta_1} a_2$.

($\Leftarrow$). By contradiction using the rule (TREE-PREC). Assuming $\exists a \in \mathsf{dom}(\theta_1)$, such that $\forall k > 0, \theta_1^k(a) \neq \theta_2(a)$, obtain $a \sqsubseteq_{\theta_2} \theta_2(a) \not\Rightarrow a \sqsubseteq_{\theta_1} \theta_2(a)$. Contradiction. □

2.2 Abstracting tree domains

Given sets $A, \hat{A}$ and a surjection $\eta : A \to \hat{A}$, one can build a Galois connection between the domains of powersets: $\langle \wp(A), \subseteq \rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \wp(\hat{A}), \subseteq \rangle$, where $\alpha(S) = \{\eta(a) \mid a \in S\}$ is an element-wise abstraction and $\gamma(\hat{S}) = \{a \mid \hat{a} \in \hat{S} \text{ and } \eta(a) = \hat{a}\}$. This Galois connection is widely employed to build families of polyvariant control-flow analysis using *context* abstractions [25]. In this case, η is referred to as an *extraction function* [18]. In this section, we assume $A, \hat{A}$ and η to be fixed and focus on building the following Galois connection: $\langle \wp(\mathcal{T}(A)), \subseteq \rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathcal{T}(\hat{A}), \sqsubseteq_{\mathcal{T}} \rangle$ for some α and γ.

Define $\alpha_{\mathcal{T}} : \mathcal{T}(A) \to \mathcal{T}(\hat{A})$ as follows:

$$\alpha_{\mathcal{T}}(\theta) = \lambda \hat{a}.\ \beta_\theta \left(\sqcup_\theta \left\{ \theta(a) \mid a \in \eta^{-1}(\hat{a}) \right\} \right), \text{where}$$

$$\beta_\theta = \lambda a. \eta \left(\sqcap_\theta \left\{ a' \in S \mid a \sqsubseteq_\theta a' \text{ and } \forall a'' \in (\eta^{-1} \circ \eta)(a') : a \sqsubseteq_\theta a'' \right\} \right)$$

The function $\alpha_{\mathcal{T}}$ works as an abstraction from the domain of *concrete* tree functions $\mathcal{T}(A)$ to the domain of *abstract* tree functions $\mathcal{T}(\hat{A})$. More precise, the θ-induced least upper bound of all immediate θ-ancestors of elements $a \in \eta^{-1}(\hat{a})$ is being abstracted via β_θ The auxiliary helper function β_θ maps an element $a \in A$ to the closest (with respect to

[3] See Wren's master's thesis for the detailed proof [26, Section 7.6].

θ-antecedence) abstract element $\hat{a} \in \hat{A}$, such that all $a' \in \eta^{-1}(\hat{a})$ non-strictly θ-antecede a. The result is expressed via a greatest lower bound $\sqcap_\theta$ since the set $\{a' \mid a \sqsubseteq_\theta a'\}$ for some fixed a is a complete ordered chain. Informally, it is a "branch" in a tree θ.

The design concern behind the definition of $\alpha_{\mathcal{T}}$ is to build an abstraction of trees that *respects* antecedence. The following lemma formalizes this result and states the inverse of $\alpha_{\mathcal{T}}$ to be *monotone* with respect to the antecedence.

Lemma 2 (Abstract antecedence). *In the stated definitions, for fixed* $A, \hat{A}, \eta : A \to \hat{A}$ *and* $\theta \in \mathcal{T}(A)$*, for all* $a_1, a_2 \in \mathsf{dom}(\theta)$*,* $\hat{a}_1 = \eta(a_1)$*,* $\hat{a}_2 = \eta(a_2)$*, such that* $\hat{a}_1 \neq \hat{a}_2$*, one has* $\hat{a}_1 \sqsubseteq_{\hat{\theta}} \hat{a}_2 \Rightarrow a_1 \sqsubseteq_\theta a_2$*, where* $\hat{\theta} = \alpha_{\mathcal{T}}(\theta)$*.*

Proof. By induction on k, such that $\hat{a}_2 = \hat{\theta}^k(\hat{a}_1)$ and the definition of $\alpha_{\mathcal{T}}$.[4] □

Lemma 3. $\alpha_{\mathcal{T}}$ *is monotone with respect to* $\sqsubseteq_{\mathcal{T}}$*.*

Since $\alpha_{\mathcal{T}}$ is proven to be monotone with respect to $\sqsubseteq_{\mathcal{T}}$, it is tempting now to state the Galois connection between concrete and abstract trees $\langle \mathcal{T}(A), \sqsubseteq_{\mathcal{T}} \rangle \underset{\alpha_{\mathcal{T}}}{\overset{\gamma_{\mathcal{T}}}{\leftrightarrows}} \langle \mathcal{T}(\hat{A}), \sqsubseteq_{\mathcal{T}} \rangle$, where $\gamma_{\mathcal{T}}(\hat{\theta}) = \sqcup_{\mathcal{T}} \left\{ \theta \mid \alpha_{\mathcal{T}}(\theta) \sqsubseteq_{\mathcal{T}} \hat{\theta} \right\}$ and employ it directly for the analysis. However, one can see that such connection does not bring much benefit because of loss of information when computing a concretization $\gamma_{\mathcal{T}}$ by merging concrete trees, i.e., using $\sqcup_{\mathcal{T}}$ for concretization is very inaccurate. Instead, we build the Galois connection between a powerset over a concrete tree domain and an abstract tree domain. Intuitively, it's preferable to merge abstract trees once all information about antecedence is abstracted, rather than to merge various concrete trees, retrieved from an abstract one.

The resulting Galois connection between $\wp(\mathcal{T}(A))$ and $\mathcal{T}(\hat{A})$ is built via functions $\alpha^\# : \wp(\mathcal{T}(A)) \to \mathcal{T}(\hat{A})$ and $\gamma^\# : \mathcal{T}(\hat{A}) \to \wp(\mathcal{T}(A))$, defined as follows:

$$\alpha^\#(S) = \bigsqcup_{\mathcal{T}} \{\alpha_{\mathcal{T}}(\theta) \mid \theta \in S\} \qquad \gamma^\#(\hat{\theta}) = \left\{ \theta \mid \alpha_{\mathcal{T}}(\theta) \sqsubseteq_{\mathcal{T}} \hat{\theta} \right\}$$

Lemma 4. *For fixed sets* A*,* $\hat{A}$ *and a surjection* $\eta : A \to \hat{A}$*, the functions* $\alpha^\#$ *and* $\gamma^\#$ *form a Galois connection* $\langle \wp(\mathcal{T}(A)), \subseteq \rangle \underset{\alpha^\#}{\overset{\gamma^\#}{\leftrightarrows}} \langle \mathcal{T}(\hat{A}), \sqsubseteq_{\mathcal{T}} \rangle$*.*

The last theorem of this section connects the antecedence in the abstract tree function $\hat{\theta}$ and its concrete counterpart θ by giving an interpretation of the abstract result.

Theorem 1 (Sound approximation of antecedence). *For fixed sets* A*,* $\hat{A}$*, a surjection* $\eta : A \to \hat{A}$*, and a Galois connection* $\langle \wp(\mathcal{T}(A)), \subseteq \rangle \underset{\alpha^\#}{\overset{\gamma^\#}{\leftrightarrows}} \langle \mathcal{T}(\hat{A}), \sqsubseteq_{\mathcal{T}} \rangle$*, given an abstract tree function* $\hat{\theta} \in \mathcal{T}(\hat{A})$*, and a concrete one* $\theta \in \mathcal{T}(A)$*. Then* $\theta \in \gamma^\#(\hat{\theta})$ *iff*

(a) $\eta(\mathsf{dom}(\theta)) \subseteq \mathsf{dom}(\hat{\theta})$ *and*
(b) for all $\hat{a}_1, \hat{a}_2 \in \mathsf{dom}(\hat{\theta})$*, such that* $\hat{a}_1 \neq \hat{a}_2$ *and* $\hat{a}_1 \sqsubseteq_{\hat{\theta}} \hat{a}_2$*, for all* $a_1 \in \eta^{-1}(\hat{a}_1) \cap \mathsf{dom}(\theta)$*,* $a_2 \in \eta^{-1}(\hat{a}_2) \cap \mathsf{dom}(\theta)$*, one has* $a_1 \sqsubseteq_\theta a_2$*.*

Proof. ($\Rightarrow$): The proof is using Lemma 2 and the definition of $\gamma^\#$. ($\Leftarrow$): The proof is by definition of $\alpha_{\mathcal{T}}$ and the optimality of $\alpha_{\mathcal{T}}(\theta)$ with respect to $\hat{\theta}$ using Lemma 1 and monotonicity of $\alpha_{\mathcal{T}}$ by Lemma 3, i.e., $\alpha_{\mathcal{T}}(\theta) \sqsubseteq_{\mathcal{T}} \hat{\theta}$. □

[4] For full proofs, see Appendix A.

2.3 Uniqueness domain

One disadvantage of the concretization function $\gamma_{\mathcal{T}}$ is its imprecision with respect to the evolution of the abstract tree. For instance, given an abstract tree function $\hat{\theta}$ and $\hat{a} \in \mathsf{dom}(\hat{\theta})$. Assume, one wants to attach a new element a to all the concretizations of $\hat{\theta}$ and then compute the abstraction following the strategy of "pushing alphas" to compute an abstract transition function [17], i.e.,

$$\alpha_{\mathcal{T}}(\{\theta[a \mapsto a'] \mid \text{for some } a' \in \eta^{-1}(\hat{a}) \text{ and } \theta \in \gamma_{\mathcal{T}}(\hat{\theta})\}).$$

Since there might be multiple different "parents" $a' \in \eta^{-1}(\hat{a})$, the result of the abstraction will certainly erase the fact of θ-antecedence of a concrete a by some concrete a'. In this situation the information about *uniqueness* of a' could be helpful. In other words, if *for all* $\theta \in \gamma_{\mathcal{T}}(\hat{\theta})$ there exists a unique $a' \in \eta^{-1}(\hat{a})$, then after updating *each* $\theta \in \gamma_{\mathcal{T}}(\hat{\theta})$, the antecedence will be preserved in the abstract result.

This brings us to the idea of an abstract domain that helps to keep the track of *unique* elements of A serving as nodes of the concrete θ with respect to the surjection η. The necessary technique can be adapted from the studies on shape analysis [13].

We define the *uniqueness* domain as a three-point partially-ordered set $\mathscr{U} = \{\bot, 1, +\}$, such that $\bot \sqsubseteq 1 \sqsubseteq +$. We will use the following notation for total functions having A as their domain and $\mathscr{U}$ as their codomain: $\mathcal{U}(A) \triangleq \langle A \to \mathscr{U}, \dot{\sqsubseteq}\rangle$, where the partial order $\dot{\sqsubseteq}$ is defined point-wise and helps to track cardinality of the set of objects, associated with some $a \in A$.

For a fixed surjection η, we build the following composition of Galois connections between sets of tree functions $\wp(\mathcal{T}(A))$ over a concrete set A and uniqueness functions $\mathcal{U}(\hat{A})$ over an abstract set $\hat{A} = \eta(A)$:

$$\langle \wp(\mathcal{T}(A)), \subseteq\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \wp(\mathcal{U}(A)), \subseteq\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \wp(\mathcal{U}(\hat{A})), \subseteq\rangle \underset{\alpha_3}{\overset{\gamma_3}{\leftrightarrows}} \langle \mathcal{U}(\hat{A}), \dot{\sqsubseteq}\rangle,$$

$$\text{where} \quad \begin{aligned} \alpha_1(S) &= \{\lambda a.\mathsf{if}\ (a \in \mathsf{dom}(\theta))\ \mathsf{then}\ 1\ \mathsf{else}\ \bot \mid \theta \in S\} \\ \gamma_1(\hat{S}) &= \bigcup\{S' \mid \alpha_1(S') \subseteq \hat{S}\} \end{aligned}$$

$$\begin{aligned} \alpha_2(S) &= \left\{\lambda \hat{a}. \biguplus_{a \in \eta^{-1}(\hat{a})} u(a) \mid u \in S\right\} \\ \gamma_2(\hat{S}) &= \bigcup\{S' \mid \alpha_2(S') \subseteq \hat{S}\} \end{aligned} \qquad \begin{aligned} \alpha_3(S) &= \lambda \hat{a}. \bigsqcup_{u \in S} u(\hat{a}) \\ \gamma_3(\hat{S}) &= \bigcup\{S' \mid \alpha_2(S') \dot{\sqsubseteq} \hat{S}\} \end{aligned}$$

The symmetric binary operation $\uplus \subseteq \mathscr{U} \times \mathscr{U}$ is defined as follows: $\bot \uplus \bot = \bot$, $1 \uplus \bot = \bot \uplus 1 = 1$, $1 \uplus 1 = +$, and for all $e \in \mathscr{U}$, $+ \uplus e = e \uplus + = +$.

Informally, the abstraction function α_1 maps a tree function θ into a uniqueness function u in a straightforward way. The function α_2 collects information about unique elements in u with respect to the surjection η. Finally, α_3 computes the *maximal* abstract number of occurrences of every abstract element among the set of abstract uniqueness functions $u \in S$ by taking a least upper bound $\bigsqcup_{u \in S}$. We define $\alpha^\flat = \alpha_3 \circ \alpha_2 \circ \alpha_1$ and $\gamma^\flat = \gamma_1 \circ \gamma_2 \circ \gamma_3$. The following theorem states the main property of the Galois connection $\langle \wp(\mathcal{T}(A)), \subseteq\rangle \underset{\alpha^\flat}{\overset{\gamma^\flat}{\leftrightarrows}} \langle \mathcal{U}(\hat{A}), \dot{\sqsubseteq}\rangle$:

Theorem 2 (Sound approximation of uniqueness in trees). *For fixed sets A, $\hat{A}$, a surjection $\eta : A \to \hat{A}$, and a Galois connection $\langle \wp(\mathcal{T}(A)), \subseteq \rangle \underset{\alpha^\flat}{\overset{\gamma^\flat}{\leftrightarrows}} \langle \mathcal{U}(\hat{A}), \dot{\sqsubseteq} \rangle$, given an abstract uniqueness function $u \in \mathcal{U}(\hat{A})$, $\theta \in \gamma^\flat(u)$, then for all $\hat{a} \in \hat{A}$:*

(a) $u(\hat{a}) = \bot$ implies for all $a \in \eta^{-1}(\hat{a})$, $a \notin \mathsf{dom}(\theta)$;
(b) $u(\hat{a}) = 1$ implies there exists at most one $a \in \eta^{-1}(\hat{a})$, such that $a \in \mathsf{dom}(\theta)$.

2.4 Reduced product

Following Cousot and Cousot's cookbook on compositional design of Galois connections [8], we use a reduced product of abstract tree and uniqueness domains in order to improve the abstraction of the concrete tree domain.

$$\langle \wp(\mathcal{T}(A)), \subseteq \rangle \underset{\alpha^\natural}{\overset{\gamma^\natural}{\leftrightarrows}} \langle \mathcal{T}(\hat{A}), \sqsubseteq_{\mathcal{T}} \rangle \times \langle \mathcal{U}(\hat{A}), \dot{\sqsubseteq} \rangle, \text{ where}$$
$$\alpha^\natural(S) = \langle \alpha^\#(S), \alpha^\flat(S) \rangle \qquad \gamma^\natural(\langle \hat{\theta}, u \rangle) = \gamma^\#(\hat{\theta}) \cap \gamma^\flat(u)$$

Intuitively, by tracking the cardinality of nodes in a concrete tree θ, we can recompute effectively an abstraction $\alpha_{\mathcal{T}}(\theta)$, just by picking the closest *uniquely-abstracted* nodes as abstract ancestors. This observation will be employed when building an abstract transition function of the dominance analysis in Section 4.

3 FJO: a Core Calculus for Java with Ownership Hierarchies

In this section, we describe A-Normal Featherweight Java with Ownership (FJO)—a simplified language, based on Featherweight Java [14] and JOE_1 [2] and aimed to to model key aspects of Java-like programming languages extended with ownership parameters in order to preserve the owners-as-dominators invariant. We also provide the definition of the concrete semantics of FJO, which, being abstracted in Section 4, gives rise to the dominance analysis.

3.1 Syntax

Figure 2 provides the definition of the syntax of FJO. To specify the ownership relation between objects, the language features *implicit ownership parametrization*, i.e., each class carries an implicit parameter, which stands for the owner of a particular instance of this class and is instantiated when a new object instance is created.

Types in FJO are different from the traditional Featherweight Java in the sense that they are parameterized with an ownership parameter p. For the sake of simplicity we restrict the number of ownership parameters to one, which is referred to inside the body of a particular class as owner.[5] Alternatively, one can specify an owner of a new

[5] However, more expressive possibilities exist in the literature, for example, by allowing the programmer to declare an arbitrary number of ownership class parameters and the expected relationship between owner parameters of a class [3].

$$
\begin{aligned}
P \in \mathsf{Program} &::= \overrightarrow{C}\ ;\ e \\
C \in \mathsf{Class} &::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{t\ f}\ ;\ \overrightarrow{M}\} \\
M \in \mathsf{Method} &::= t\ m(t\ v)\ \{\overrightarrow{t\ v}\ ;\ \overrightarrow{s}\} \\
p \in \mathsf{Owner} &::= \mathsf{this} \mid \mathsf{owner} \mid \mathsf{root} \mid v
\end{aligned}
\qquad
\begin{aligned}
t \in \mathsf{Type} &::= C\langle p\rangle \\
s \in \mathsf{Stmt} &::= v = e;^{\ell} \mid v.f = v;^{\ell} \mid \\
&\quad\ \ \texttt{return}\ v;^{\ell} \\
e \in \mathsf{Exp} &::= v \mid v.f \mid \texttt{new}\ t \mid v.m(v)
\end{aligned}
$$

$C \in \mathsf{ClassName}$ is a set of class names
$v \in \mathsf{Var}$ is a set of variable names
$f \in \mathsf{FName}$ is a set of field names
$m \in \mathsf{MName}$ is a set of method names
$\ell \in \mathsf{Lab}$ is a set of labels

Fig. 2: Syntax of A-Normal Featherweight Java with Ownership

instances using this-reference, the reference to the global object root or some local variable v. When a new object $\texttt{new}\ C\langle p\rangle$ is allocated, its owner p should be specified explicitly. This enables a tree of owners to be maintained at runtime.

Type soundness and confinement preservation theorems ensure that the OAD invariant is not violated during the program execution [4, 2]. We omit the typing rules for the sake of brevity, and instead refer the interested reader to the third author's PhD dissertation [3].

3.2 A concrete state space of FJO

The concrete state space of execution of FJO (Figure 3) is similar to the formalism of Smaragdakis et al. [24]. Every concrete state of the corresponding abstract machine contains a local environment B, a shared store σ and a tree function θ, which describes the ownership tree, as its components. The state space is parametrized by the sets of *contexts* and *heap contexts*. Contexts are used by a store to maintain information about local variables of methods and serialized continuations. Heap contexts are used to store values of objects fields. The purpose of the contexts is to introduce an extra level of indirection through a store, which makes it easy to collapse information about different

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Stmt} \times \mathit{Env} \times \mathit{Store} \times \mathit{ContSensAddr} \times \mathit{Context} \times \mathit{Tree} \\
B \in \mathit{Env} &= (\mathsf{Var} \rightharpoonup \mathit{ContSensAddr}) \times \mathit{HContext} \\
\sigma \in \mathit{Store} &= \mathit{ContSensAddr} \rightharpoonup (\mathit{Obj} + \mathit{Continuation}) \\
o \in \mathit{Obj} &= \mathit{HContext} \times (\mathsf{FName} \rightharpoonup \mathit{ContSensAddr}) \\
k \in \mathit{Continuation} &= (\mathit{ContSensAddr} \times \mathit{Env} \times \mathsf{Stmt} \times \mathit{Context} \times \mathit{ContSensAddr}) \\
a \in \mathit{ContSensAddr} &= (\mathsf{Var} \times \mathit{Context}) + (\mathsf{FName} \times \mathit{HContext}) + (\mathsf{MName} \times \mathit{Context}) \\
\theta \in \mathit{Tree} &= \mathcal{T}(\mathit{HContext}) \\
c \in \mathit{Context} &\text{ is an infinite set of contexts} \\
hc \in \mathit{HContext} &\text{ is an infinite set of heap contexts } \cup \{\mathsf{root}\}
\end{aligned}
$$

Fig. 3: A concrete state-space of A-Normal Featherweight Java with Ownership

program paths just by limiting the sets *Context* and *HContext*. Both these sets are left unspecified to make context-sensitivity an external parameter.

In the described settings, heap contexts play the role of *object identifiers*. In the concrete semantics, one should chose the strategy of heap context allocation in a way that every new object has a unique heap context, so each object is represented by its heap context and a map of fields. The tree domain, *Tree*, is built on top of the set of heap contexts combined with root, which is assumed to be the root of all trees from *Tree*. When a method is invoked, the corresponding continuation is allocated in the store. A continuation consist of the return address, the callee's environment, the next statement to be executed, the callee's local context and the address of the previous serialized continuation. Storing continuations in the same store as objects gives an easy way to eliminate infinite recursive calls when building a context-sensitive analysis. Finally, each local environment B is a tuple that consists of a map from local variables to store addresses and an identifier of an object, referred by the current this-reference.

3.3 A concrete semantics of FJO

The runtime semantics of FJO is expressed using a small-step CEK machine [9], defined by the transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. We assume each statement and the program has a unique label ℓ, and the helper function *succ* returns the subsequent statement for a statements label. The semantics is parametrized by two helper functions that define context allocating strategies: *merge* and *record* [24]. We omit most of the transition rules as they are not relevant to the computation of ownership trees and provide the concrete semantics for the only essential rule, object instance allocation:[6]

$$\left[\begin{array}{ccc} \multicolumn{3}{c}{([\![v = \texttt{new}\ C\langle p\rangle;^{\ell}]\!], B, \sigma, a_k, c, \theta) \Rightarrow (succ(\ell), B, \sigma', a_k, c, \theta'), \text{ where}} \\ hc = record(\ell, c) & \overrightarrow{f} = \mathcal{F}(C) & a_i = (f_i, hc) \\ o' = (hc, [f_i \mapsto a_i]) & \sigma' = \sigma + [B(v) \mapsto o'] & \theta' = adjust(\theta, p, hc, B, \sigma) \end{array}\right]$$

Informally, each *new* object creation in the concrete execution semantics will increase the tree by attaching a new node to it. The possible growth of the tree is not limited, because the set of heap contexts *HContext* is considered infinite. The function *adjust*, responsible for maintaining the tree of owners θ, is crucial in our semantics:

$$adjust(\theta, p, hc, B, \sigma) = \theta[hc \mapsto hc'], \text{where}$$

$$hc' = \begin{cases} \mathsf{root} & \text{if } p = \mathsf{root} \\ \pi_2(B) & \text{if } p = \mathsf{this} \\ \theta(\pi_2(B)) & \text{if } p = \mathsf{owner} \\ hc'', \text{ where } (hc'', _) = \sigma(B(v)) & \text{if } p = v \text{ for some } v \in \mathsf{Var} \end{cases}$$

Depending on the syntactic form of the specified owner p, an owner is assigned to a newly allocated object with an id hc. If $p = \mathsf{root}$, the owner is obviously root. If $p = \mathsf{this}$, the owner is retrieved as a second projection π_2 of the local environment B,

[6] Full description of the concrete semantics is provided in Appendix C.

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Stmt} \times \widehat{Env} \times \widehat{Store} \times \widehat{ContSensAddr} \times \widehat{Context} \times \widehat{Tree} \\
\hat{B} \in \widehat{Env} &= (\mathsf{Var} \rightharpoonup \widehat{ContSensAddr}) \times \widehat{HContext} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{ContSensAddr} \rightharpoonup (\wp(\widehat{Obj}) + \wp(\widehat{Continuation})) \\
\hat{o} \in \widehat{Obj} &= \widehat{HContext} \times (\mathsf{FName} \rightharpoonup \widehat{ContSensAddr}) \\
\hat{k} \in \widehat{Continuation} &= (\widehat{ContSensAddr} \times \widehat{Env} \times \mathsf{Stmt} \times \widehat{Context} \times \widehat{ContSensAddr}) \\
\hat{a} \in \widehat{ContSensAddr} &= (\mathsf{Var} \times \widehat{Context}) + (\mathsf{FName} \times \widehat{HContext}) + (\mathsf{MName} \times \widehat{Context}) \\
\hat{\theta}_u \in \widehat{Tree} &= \mathcal{T}(\widehat{HContext}) \times \mathcal{U}(\widehat{HContext}) \\
\hat{c} \in \widehat{Context} &\text{ is a finite set of contexts} \\
\hat{hc} \in \widehat{HContext} &\text{ is a finite set of heap contexts } \cup \{\mathsf{root}\}
\end{aligned}
$$

Fig. 4: Abstract state-space of A-Normal Featherweight Java with Ownership for a context-sensitive dominance analysis

i.e., the current id of an object, referred to by this-reference. In the case $p = \mathsf{owner}$, the owner is taken from the tree θ as an immediate θ-ancestor of this. Finally, if an owner is specified by a variable v, it's retrieved using environment B and a store σ.

It is important to notice that the described semantics accounting to ownership trees is not supposed be implemented in the real compiler. In the original work on ownership types, an ownership tree is a purely *virtual* concept. The only purpose of having the tree as a component in the instrumented concrete state space is to build a static analysis by a systematic abstraction of a concrete semantics.

4 Abstract Semantics of FJO and Dominance Analysis

In this section, we convert the concrete semantics of FJO into an abstract interpretation of itself using the recipe of Van Horn and Might [25]. The abstraction over contexts and heap contexts is determined by an opaque surjection η, which accounts for polyvariance. Figure 4 provides a definition of the abstract state-space, which reflects its concrete counterpart and is obtained via a structural abstraction map (Figure 5). The partial order on domains of the state-space is obtained by the natural point-wise and element-wise lifting of the original partial orders. The only interesting detail to notice is that the concrete tree domain $Tree = \mathcal{T}(HContext)$ is being abstracted using the reduced product from Section 2.4 via $\alpha^{\natural} : \mathcal{T}(HContext) \to \mathcal{T}(\widehat{HContext}) \times \mathcal{U}(\widehat{HContext})$.

The abstract semantics is encoded as a small-step nondeterministic transition relation $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$. The operation $\sqcup$ for stores merges the sets for the same key value. Again, we provide only a selected rule for object allocation:[7]

$$
\left[\begin{array}{lll}
\multicolumn{3}{c}{(\llbracket v = \mathtt{new}\ C\langle p\rangle;^{\ell}\rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \leadsto (succ(\ell), \hat{B}, \hat{\sigma}', \hat{a}_k, \hat{c}, \hat{\theta}'_{u'}), \text{ where}} \\
\hat{hc} = \widehat{record}(\ell, \hat{c}) & \overrightarrow{f} = \mathcal{F}(C) & \hat{a}_i = (f_i, \hat{hc}) \\
\hat{o}' = (\hat{hc}, [f_i \mapsto \hat{a}_i]) & \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{B}(v) \mapsto \hat{o}'] & \hat{\theta}'_{u'} = \widehat{adjust}(\hat{\theta}_u, p, \hat{hc}, \hat{B}, \hat{\sigma})
\end{array}\right]
$$

[7] The rest of abstract semantics is provided in Appendix C.

$$
\begin{aligned}
\alpha_\Sigma(S) &= \{\alpha_{State}(s,B,\sigma,a_k,c,\theta) \mid (s,B,\sigma,a_k,c,\theta) \in S\} \\
\alpha_{State}(s,B,\sigma,a_k,c,\theta) &= (s,\alpha_{Env}(B),\alpha_{Store}(\sigma),\eta(a_k),\eta(c),\alpha^\natural(\theta)) \\
\alpha_{Env}(B) &= \lambda v.\,\eta(B(v)) \\
\alpha_{Store}(\sigma) &= \lambda \hat{a}.\,\{\alpha_D(\sigma(a)) \mid \eta(a) = \hat{a}\} \\
\alpha_D(a_{ret},B,s,c,a_k) &= (\eta(a_{ret}),\alpha_{Env}(B),s,\eta(c),\eta(a_k)) \\
\alpha_D(hc,[f \mapsto a_f]) &= (\eta(hc),[f \mapsto \eta(a_f)]) \\
\eta &\text{ is defined by polyvariance}
\end{aligned}
$$

Fig. 5: A structural abstraction map

The abstracted function $\widehat{adjust}$ merges abstract ownership trees in a way that respects antecedence in the concrete counterpart. In order to define it, we need introduce an auxiliary function to compute the least unique $\hat{\theta}$-ancestor:

$$
unique^u_{\hat{\theta}}(\hat{hc}) \triangleq \hat{hc}', \text{where}
$$

$\hat{hc} \sqsubseteq_{\hat{\theta}} \hat{hc}'$, $u(\widehat{hc'}) = 1$ and for all $\hat{hc}''$, $\hat{hc} \sqsubseteq_{\hat{\theta}} \hat{hc}'' \Rightarrow \hat{hc}' \sqsubseteq_{\hat{\theta}} \hat{hc}''$.

The abstracted version of $\widehat{adjust}$ is defined as follows:

$$
\widehat{adjust}(\hat{\theta}_u, p, \hat{hc}, \hat{B}, \hat{\sigma}) = \hat{\theta}''_{u'}, \text{ where}
$$

$$
\begin{aligned}
&(\hat{\theta}, u) = \hat{\theta}_u \\
&u' = u[\hat{hc} \mapsto u(\hat{hc}) \uplus 1] \\
&\hat{hc}_i = children_{\hat{\theta}}(\hat{hc}) \\
&\hat{\theta}' = \hat{\theta}[\hat{hc}_i \mapsto \hat{\theta}(\hat{hc})] \\
&\hat{\theta}'' = \textstyle\bigsqcup_{\mathcal{T}}(\hat{\theta}_j \cup \{\hat{\theta}'\})
\end{aligned}
\qquad
\hat{\theta}_j = \begin{cases}
\hat{\theta}[\hat{hc} \mapsto \mathsf{root}] \text{ if } p = \mathsf{root} \\
\hat{\theta}[\hat{hc} \mapsto unique^{u'}_{\hat{\theta}}(\pi_2(\hat{B}))] \text{ if } p = \mathsf{this} \\
\hat{\theta}[\hat{hc} \mapsto unique^{u'}_{\hat{\theta}}(\hat{\theta}(\pi_2(\hat{B})))] \text{ if } p = \mathsf{owner} \\
\left\{\hat{\theta}[\hat{hc} \mapsto unique^{u'}_{\hat{\theta}}(\hat{hc}')] \mid (\hat{hc}', _) \in \hat{\sigma}(\hat{B}(v))\right\} \text{ if } p = v
\end{cases}
$$

If a node $\hat{hc}$ is already present in the abstract tree, attaching of it as a leaf might violate the antecedence in the concrete tree. For this purpose, all *children* of $\hat{hc}$ in the abstract tree are "reattached" to the $\hat{\theta}$-ancestor of $\hat{hc}$. Lemma 5 formalizes the observation that only heap contexts, whose pre-images are unique in the concretization with respect to η, may serve as non-leaf nodes in an abstract tree function.

Lemma 5. *Define the invariant $\mathscr{I}(\hat{\theta}_u)$ as follows:*

$$
\mathscr{I}(\hat{\theta}_u) \triangleq \forall \hat{a} \in \mathsf{dom}(\hat{\theta}) : children_{\hat{\theta}}(\hat{a}) \neq \emptyset \Rightarrow u(\hat{a}) = 1.
$$

Then $\mathscr{I}(\hat{\theta}_u)$ and $\hat{\theta}'_{u'} = \widehat{adjust}(\hat{\theta}_u, p, \hat{hc}, \hat{B}, \hat{\sigma})$ imply $\mathscr{I}(\hat{\theta}'_{u'})$

Proof. The proof is by the definition of $\widehat{adjust}$ and case analysis on p. □

We assume the stated invariant to be held for the initial abstract state $\hat{\varsigma}_0$.

Termination and complexity The state space is implemented following the methodology of Might [18], so its dependence structure is a directed acyclic graph. All leaf nodes of this graph correspond to finite sets thanks to the chosen extraction function η, so the whole state space is finite. One can consider an abstract collecting semantics as an iteration over a set of reachable abstract states $\hat{\mathscr{F}} : \wp(\hat{\Sigma}) \to \wp(\hat{\Sigma})$:

$$\hat{\mathscr{F}}(\hat{S}) = \{\hat{\varsigma}_0\} \cup \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{S} \text{ and } \hat{\varsigma} \leadsto \hat{\varsigma}'\},$$

where $\hat{\varsigma}_0$ is an initial abstract state. By Kleene's fixed point theorem there exists $n \in \mathbb{N}$, such that $lfp(\hat{\mathscr{F}}) = \hat{\mathscr{F}}^n(\emptyset)$.

To estimate the complexity of the analysis (with η implementing Shivers's *time*-based k-CFA) we apply the single-threaded store optimization [19] and a similar single-threaded tree-uniqueness optimization. In these settings, $|\widehat{Tree}| = O(|\mathsf{Lab}|^{3k}) = O(n^{3k})$, where n is the size of the program and k is k-CFA's polyvariance parameter. Hence for a fixed k the overall complexity remains polynomial in the size of the program.

Soundness The soundness of the analysis is stated by the following theorem:

Theorem 3 (Soundness of the dominance analysis). *If* $\alpha_{State}(\varsigma) \sqsubseteq \hat{\varsigma}$ *and* $\varsigma \Rightarrow \varsigma'$, *then there must exist a state* $\hat{\varsigma}'$, *such that* $\alpha_{State}(\varsigma') \sqsubseteq \hat{\varsigma}'$ *and* $\hat{\varsigma} \leadsto \hat{\varsigma}'$.

Proof. (Sketch) The proof is a standard one by the case-analysis of the concrete transition relation ($\Rightarrow$). The only non-trivial case is an object allocation. For this particular case the soundness if reformulated in terms of concretization $\gamma^{\natural}$ of an abstracted tree function and proven separately for both abstract domains $\mathcal{T}(\widehat{HContext})$ and $\mathcal{U}(\widehat{HContext})$, employing Theorem 1 and the invariant from Lemma 5.[8] $\square$

5 Discussion and Related Work

Similar analyses and formalisms We developed our work mainly by exploring two domains: of trees and uniqueness functions, combining and embedding them into Cousot and Cousot's work on abstract interpretation [7, 8]. The resulting framework is built using an adapted idea of Shivers's k-CFA [23] and systematic abstraction of the store-based semantic formalism [18, 19, 24, 25].

Our analysis has been developed independently from a recently published similar work on analyses for concurrent higher-order programs [20]. Although the later one solves a different class of problems, it employs the same sort of uniqueness abstraction to track the cardinality of allocated abstract thread identifiers. It would be interesting to investigate the impact of the dominance analysis in a concurrent setting.

Ownership and dominance inference The idea of tree domains originates in the notion of *ownership functions*, introduced by Wren is his master's thesis [26] and employed by him for ownership inference. The author describes a dynamic analysis via

[8] The detailed proof is provided in Appendix A.

runtime recomputing the dominance tree. We believe that, using abstraction of tree domains from our work, it is possible to develop a corresponding static analysis by abstract interpretation, using a function similar to our $\widehat{adjust}$. Such an analysis could utilize Alstrup and Lauridsen's dynamic algorithm for efficiently maintaining a dominator tree during the fixed point computation [1].

Our result is close to the static dominance inference by Milanova and Vitek [21]. Their work however requires a complex correctness proof. We believe that the systematic abstract interpretation approach is preferable as the close connection between the formal semantics and the analysis eases the proof burden. Also, their analysis is not tunable with respect to polyvariance. Fluet and Weeks [10] develop a *contification* analysis by employing dominance in abstract call graphs of functional programs. We believe that reformulating their analysis using tree domains would allow to tune the precision of the result. Geilmann and Poetschz-Heffter [12] developed a modular abstract interpretation analysis to discover simple (i.e., non-hierarchical) confinement properties in Java-like programs. This work employs a *box model* instead of tree domains.

Trees as an abstract domain We suspect that the tree semi-lattice underlies the efficient algorithms for dominators of control-flow graphs in the work of Cooper et al. [5], which is formulated in terms of the monotone framework. Gallagher et al. have used tree domains extensively in abstract interpretation however in the form of *tree automata* or *tree grammars* [11].

6 Conclusion and Future Work

We have presented a static analysis to determine dominance in object graphs. Existing points-to analyses are not designed to properly abstract this relation between objects. Our framework employs information, provided by ownership type annotations and operates on a product of two formalized domains: of trees and uniqueness functions. The analysis was developed by using Galois connections for systematic abstraction of a small-step abstract machine instrumented for dominance tree computations and proven to be sound. We have developed a prototype implementation of the resulting analysis in Scala along with a series of examples.[9]

As future work, we believe that employing gradual ownership types [22] will help to overcome the verbosity of the type annotations and make them a powerful tool for static analyses. For instance, the presented analysis uses only a minimal amount of annotations at allocation sites. Going in the other direction, we plan to use tree domains for building ownership inference algorithms by systematic abstraction of a dynamic semantics of dominance tree computation [1, 26]. In our formalism, classes are limited to only one ownership parameter. In future work, we plan to remove this restriction by keeping a separate map of ownership parameters for each heap context and maintain it according to the ownership tree structure to preserve the ordering [2].

It is also an open question, which context allocation strategies would yield the most precise analysis in terms of dominance trees and how one can use ownership information to improve the results of traditional points-to analyses.

[9] Available at `http://people.cs.kuleuven.be/ilya.sergey/ownership-cfa`

References

1. S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical report, University of Copenhagen, 1996.
2. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02*, pages 292–310.
3. D. G. Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, New South Wales, Australia, 2003.
4. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64.
5. K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, Rice University Houston, Texas, USA, 2001.
6. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
9. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, August 2009.
10. M. Fluet and S. Weeks. Contification using dominators. In *ICFP '01*, pages 2–13.
11. J. P. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *PADL '02*, pages 243–261.
12. K. Geilmann and A. Poetzsch-Heffter. Modular Checking of Confinement for Object-Oriented Components using Abstract Interpretation. In *IWACO '11*, 2011.
13. P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.
14. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, May 2001.
15. N. D. Jones. Flow analysis of lambda expressions (preliminary version). In *ICALP '81*, pages 114–128.
16. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1:121–141, January 1979.
17. J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *ICFP '09*, pages 287–298.
18. M. Might. Abstract interpreters for free. In *SAS '10*, pages 407–421.
19. M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10*, pages 305–315.
20. M. Might and D. Van Horn. A family of abstract interpretations for static analysis of concurrent higher-order programs. Accepted to *SAS '11*.
21. A. Milanova and J. Vitek. Static dominance inference. In *TOOLS '11*, pages 211–227.
22. I. Sergey and D. Clarke. Towards gradual ownership types. In *IWACO '11*, 2011.
23. O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.
24. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL '11*, pages 17–30.
25. D. Van Horn and M. Might. Abstracting Abstract Machines. In *ICFP '10*, pages 51–62.
26. A. Wren. Inferring ownership. Master's thesis, Imperial College London, UK, June 2003.

A Proofs of Some Statements From Sections 2 and 4

Lemma 2 (Abstract antecedence). *In the definitions of Section 2, for fixed* $A, \hat{a}, \eta : A \to \hat{a}$ *and* $\theta \in \mathcal{T}(A)$*, for all* $a_1, a_2 \in \mathsf{dom}(\theta)$*,* $\hat{a}_1 = \eta(a_1)$*,* $\hat{a}_2 = \eta(a_2)$*, such that* $\hat{a}_1 \neq \hat{a}_2$*, one has* $\hat{a}_1 \sqsubseteq_{\hat{\theta}} \hat{a}_2 \Rightarrow a_1 \sqsubseteq_\theta a_2$*, where* $\hat{\theta} = \alpha_{\mathcal{T}}(\theta)$*.*

Proof. By induction on k, such that $\hat{a}_2 = \hat{\theta}^k(\hat{a}_1)$ and the definition of $\beta_{\mathcal{T}}$.

Case $k = 1$**.** By the definition of tree functions,

$$\hat{a}_2 = \hat{\theta}(\hat{a}_1) = \beta_\theta \left(\sqcup_\theta \underbrace{\left\{\theta(a) \mid a \in \eta^{-1}(\hat{a}_1)\right\}}_{(\zeta)} \right)$$

By assumption $a_1 \in \zeta$, which implies $a_1 \sqsubseteq_\theta a'$ (a), where $a' = \sqcup_\theta \zeta$.
Taking $\hat{a}_2 = \beta_\theta(a')$, we obtain $\hat{a}_2 = \eta(\theta^k(a'))$ for some $k \geq 0$. Moreover, by assumptions $a_2 \in \eta^{-1}(\hat{a}_2)$ and $\theta^k(a') \in \eta^{-1}(\hat{a}_2)$, so by the definition of β_θ, we have $a' \sqsubseteq_\theta a_2$ (b).
Using transitivity of $\sqsubseteq_\theta$, combining of (a) and (b) conclude the proof for this case.

Case $k > 1$**.** Take $\hat{a}' = \hat{\theta}(\hat{a}_1)$. Then for all $a' \in \eta^{-1}(\hat{a}') : a_1 \sqsubseteq_\theta a'$. Using induction hypothesis for any taken a' and a_2 concludes the proof.

□

Lemma 3. $\alpha_{\mathcal{T}}$ *is monotone with respect to* $\sqsubseteq_{\mathcal{T}}$*.*

Proof. Assume, there are θ_1, θ_2, $\hat{\theta}_1 = \alpha_{\mathcal{T}}(\theta_1)$ and $\hat{\theta}_2 = \alpha_{\mathcal{T}}(\theta_2)$, such that $\theta_1 \sqsubseteq_{\mathcal{T}} \theta_2$ and $\hat{\theta}_1 \not\sqsubseteq_{\mathcal{T}} \hat{\theta}_2$. Then by Lemma 1, there must exist $\hat{a}_1, \hat{a}_2 \in \mathsf{dom}(\hat{\theta}_1) \cap \mathsf{dom}(\hat{\theta}_2)$, $\hat{a}_1 \neq \hat{a}_2$, such that

(a) $\hat{a}_1 \sqsubseteq_{\hat{\theta}_2} \hat{a}_2$;
(b) $\hat{a}_1 \not\sqsubseteq_{\hat{\theta}_2} \hat{a}_1$.

From (a), using Lemma 2, conclude: $\forall a_1 \in \eta^{-1}(\hat{a}_1) \cap \mathsf{dom}(\theta_1)$, $a_2 \in \eta^{-1}(\hat{a}_2) \cap \mathsf{dom}(\theta_1)$, (c) $a_1 \sqsubseteq_{\theta_2} a_2$, which implies, by Lemma 1, (d) $a_1 \sqsubseteq_{\theta_1} a_2$. Employing the fact that $\mathsf{dom}(\theta_1) \subseteq \mathsf{dom}(\theta_2)$ and the definition of $\alpha_{\mathcal{T}}$, one can see, that $\hat{a}_1 \not\sqsubseteq_{\hat{\theta}_1} \hat{a}_2 \Leftrightarrow \exists a'_2 \in \eta^{-1}(\hat{a}_2) \cap \mathsf{dom}(\theta_1)$, such that $a_1 \not\sqsubseteq_{\theta_1} a_2$, but this contradicts (d). A contradiction proves $\hat{\theta}_1 \sqsubseteq_{\mathcal{T}} \hat{\theta}_2$. □

Lemma 4. *For fixed sets A,* $\hat{A}$ *and a surjection* $\eta : A \to \hat{A}$*, the functions* $\alpha^\#$ *and* $\gamma^\#$ *form a Galois connection*

$$\langle \wp(\mathcal{T}(A)), \subseteq \rangle \underset{\alpha^\#}{\overset{\gamma^\#}{\leftrightarrows}} \langle \mathcal{T}(\hat{A}), \sqsubseteq_{\mathcal{T}} \rangle, \text{where}$$

$$\alpha^\#(S) = \sqcup_{\mathcal{T}} \{\alpha_{\mathcal{T}}(\theta) \mid \theta \in S\}$$
$$\gamma^\#(\hat{\theta}) = \left\{\theta \mid \alpha_{\mathcal{T}}(\theta) \sqsubseteq_{\mathcal{T}} \hat{\theta}\right\}.$$

Proof. The proof of using the traditional characteristics of a Galois connection for the statement of the lemma:

$$\forall S \in \wp(\mathcal{T}(A)), \hat{\theta} \in \mathcal{T}(\hat{A}) : \alpha^{\#}(S) \sqsubseteq_{\mathcal{T}} \hat{\theta} \Longleftrightarrow S \subseteq \gamma^{\#}(\hat{\theta}).$$

$\Rightarrow$ Assume $\alpha^{\#}(S) \sqsubseteq_{\mathcal{T}} \hat{\theta}$, but $S \not\subseteq \gamma^{\#}(\hat{\theta})$. Then, by the definition of $\gamma^{\#}$, there must exist $\theta' \in S$, such that $\alpha_{\mathcal{T}}(\theta') \not\sqsubseteq_{\mathcal{T}} \hat{\theta}$. But by the definition of $\alpha^{\#}$,

$$\hat{\theta}'' = \sqcup_{\mathcal{T}} \{\alpha_{\mathcal{T}}(\theta) \mid \theta \in S\} \;\; \sqsubseteq_{\mathcal{T}} \;\; \hat{\theta}$$

and $\alpha_{\mathcal{T}}(\theta') \sqsubseteq_{\mathcal{T}} \hat{\theta}''$, which means that $\alpha_{\mathcal{T}}(\theta') \sqsubseteq_{\mathcal{T}} \hat{\theta}$. A contradiction.

$\Leftarrow$ Assume $S \subseteq \gamma^{\#}(\hat{\theta})$, but $\alpha^{\#}(S) \not\sqsubseteq_{\mathcal{T}} \hat{\theta}$, i.e., there exists $\theta' \in S$, such that $\alpha_{\mathcal{T}}(\theta') \not\sqsubseteq_{\mathcal{T}} \hat{\theta}$. But by the definition of $\gamma^{\#}$,

$$S \subseteq \left\{\theta \mid \alpha_{\mathcal{T}}(\theta) \sqsubseteq_{\mathcal{T}} \hat{\theta}\right\},$$

which implies that $\alpha_{\mathcal{T}}(\theta') \sqsubseteq_{\mathcal{T}} \hat{\theta}$. A contradiction completes the proof.

□

Theorem 3 (Soundness of the antecedence analysis). *If* $\alpha_{State}(\varsigma) \sqsubseteq \hat{\varsigma}$ *and* $\varsigma \Rightarrow \varsigma'$, *then there must exist a state* $\hat{\varsigma}'$*, such that* $\alpha_{State}(\varsigma') \sqsubseteq \hat{\varsigma}'$ *and* $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$.

Proof. The proof is a standard one by the case-analysis of the concrete transition relation ($\Rightarrow$) [19]. The only non-trivial case is an object allocation. For this particular case, since the abstract transition relation ($\rightsquigarrow$) is deterministic, we reformulate the statement of the theorem as follows (in the definitions of Section 4), using the definition of the Galois connection $\langle \alpha^{\natural}, \gamma^{\natural} \rangle$:

$$\left.\begin{array}{c} \theta \in \gamma^{\natural}(\hat{\theta}_u) \\ \mathscr{I}(\hat{\theta}_u) \\ \theta' = adjust(\theta, p, hc, B, \sigma) \\ \hat{\theta}'_{u'} = \widehat{adjust}(\hat{\theta}, p, \hat{hc}, \widehat{B}, \hat{\sigma}) \end{array}\right\} \Rightarrow \theta' \in \gamma^{\natural}(\hat{\theta}'_{u'}),$$

where the invariant $\mathscr{I}(\hat{\theta}_u)$ is defined in Lemma 5 and obviously preserved during the abstract execution of the program. Assume $\eta : HContext \to \widehat{HContext}$ is a fixed surjection, defined by polyvariance. Consider the abstraction componentwise:

Uniqueness domain Assume $u' = \pi_2(\widehat{adjust}(\hat{\theta}_u, p, \hat{hc}, \ldots))$ and $u'' = \alpha^{\flat}(\theta')$. Then $hc = \eta(hc)$. By the definition of $\widehat{adjust}$, $u'(\hat{hc}) = u(\hat{hc}) \uplus 1$. One can see that for all $\hat{hc}' \neq \hat{hc}$: $u'(\hat{hc}') = u''(\hat{hc}')$ and by the definition of $\alpha^{\flat}$ (Section 2.3), $u''(\hat{hc}) = \biguplus_{hc'' \in \eta^{-1}(\hat{hc})} \alpha_1(\theta) \uplus 1 \sqsubseteq u(\hat{hc}) \uplus 1 = u'(\hat{hc})$, so we have $u'' \dot{\sqsubseteq} u'$, i.e., $\alpha^{\flat}(\theta') \dot{\sqsubseteq} u'$, which is equivalent to $\theta' \in \gamma^{\flat}(u')$ by the main property of Galois connections.

Tree domain Taking $\theta \in \gamma^{\#}(\hat{\theta})$, by Theorem 1, we have for all $\hat{hc}_1, \hat{hc}_2 \in \mathsf{dom}(\hat{\theta})$, such that $\hat{hc}_1 \neq \hat{hc}_2$ and $\hat{hc}_1 \sqsubseteq_{\hat{\theta}} \hat{hc}_2$, for all $hc_1 \in \eta^{-1}(\hat{hc}_1) \cap \mathsf{dom}(\theta)$, $hc_2 \in \eta^{-1}(\hat{hc}_2) \cap \mathsf{dom}(\theta)$ imply $hc_1 \sqsubseteq_{\theta} hc_2$.

Using the same theorem, we will show $\theta' \in \gamma^{\#}(\hat{\theta}')$ as for all $\hat{hc}_1, \hat{hc}_2 \in \mathsf{dom}(\hat{\theta}')$, such that $\hat{hc}_1 \neq \hat{hc}_2$ and $\hat{hc}_1 \sqsubseteq_{\hat{\theta}'} \hat{hc}_2$, for all $hc_1 \in \eta^{-1}(\hat{hc}_1) \cap \mathsf{dom}(\theta')$, $hc_2 \in \eta^{-1}(\hat{hc}_2) \cap \mathsf{dom}(\theta')$ one has $hc_1 \sqsubseteq_{\theta'} hc_2$.

Assume $children_{\hat{\theta}}(\hat{hc}) = \emptyset$. Proof the statement of the theorem by the case analysis on p.

Case $p = \mathsf{root}$**.** The proof for this case is trivial since root is unique in any abstraction.

Case $p = \mathsf{this}$**.** Let $\hat{hc}' = \pi_1(\widehat{B})$, then by the polyvariance there must exist $hc' \in \eta^{-1}(\hat{hc}')$, such that $\theta' = \theta[hc \mapsto hc']$. By the definition of $\widehat{adjust}$, $\hat{\theta}' = \hat{\theta} \sqcup_{\mathcal{T}} \hat{\theta}[\hat{hc} \mapsto \hat{hc}'']$, such that (a) $\hat{hc}' \sqsubseteq_{\hat{\theta}} \hat{hc}''$ and (b) $u'(\hat{hc}'') = 1$.

Together, (a) and (b) imply that there must exist a *unique* $hc'' \in \mathsf{dom}(\theta')$, such that $hc' \sqsubseteq_{\theta} hc'' \Rightarrow hc \sqsubseteq_{\theta'} hc''$, i.e. $\forall hc'' \in \eta^{-1}(\hat{hc}'') : hc \sqsubseteq_{\theta'} hc''$.

All other $hc''' \in \eta^{-1}(\hat{hc})$ are handled by the least upper bound $\sqcup_{\mathcal{T}}$ in the definition of $\widehat{adjust}$. More precise, we employ the invariant $\mathscr{I}(\hat{\theta}_u)$ and Lemma 5 in the sense that there must exists a u-unique $hc_0 \in \eta^{-1}(\hat{hc}_0)$, such that $\hat{\theta}(\hat{hc}) = \hat{hc}_0$, $\hat{hc}_0 \neq \hat{hc}$ and $hc''' \sqsubseteq_{\theta} h_0$. In the resulting abstract tree $\hat{\theta}' = \hat{\theta} \sqcup_{\mathcal{T}} \hat{\theta}[\hat{hc} \mapsto \hat{hc}'']$, $\hat{\theta}'(\hat{hc}) = \hat{hc}^*$, so $\hat{hc}'' \sqsubseteq_{\hat{\theta}'} \hat{hc}^*$ and $\hat{hc}_0 \sqsubseteq_{\hat{\theta}'} \hat{hc}^*$. Therefore, by Lemma 5 and Theorem 2, $\exists!\ hc^* \in \eta^{-1}(\hat{hc}^*)$, such that for all $hc''' \in \eta^{-1}(\hat{hc}) \cap \mathsf{dom}(\theta')$, $hc''' \sqsubseteq_{\theta'} hc^*$.

For other abstract contexts $\hat{hc}''' \neq \hat{hc}$ the proof follows from the application of Theorem 1 to θ and $\hat{\theta}$, since $children_{\hat{\theta}}(\hat{hc}) = \emptyset$ by assumption.

Case $p = \mathsf{owner}$**.** Assuming $\hat{hc}' = \hat{\theta}(\pi_1(\widehat{B}))$ and $hc' \in \eta^{-1}(\hat{hc}')$, consider the *unique* $hc'' \in \eta^{-1}(\hat{hc}'')$, where $\hat{hc}'' = unique^{u'}_{\hat{\theta}}(\hat{hc}')$, such that $hc \sqsubseteq_{\theta'} hc' \sqsubseteq_{\theta'} hc''$. The rest of the proof is similar to the case $p = \mathsf{this}$.

Case $p = v$**.** The proof for this case is similar to the two previous cases. The only difference is that the least upper bound is computed, taking into account all possible abstract ancestors of an object, referred by v, so the resulting abstract tree function is $\sqsubseteq_{\mathcal{T}}$-greater than all of them, i.e., less precise and more general.

Now assume $\hat{hc}_i = children_{\hat{\theta}}(\hat{hc}) \neq \emptyset$. Then $\hat{\theta} \sqsubseteq_{\mathcal{T}} \hat{\theta}^* = \hat{\theta}[\hat{hc}_i \mapsto \hat{\theta}(\hat{hc})]$. As in the case $p = \mathsf{this}$, we employ the invariant $\mathscr{I}(\hat{\theta}_u)$ and Lemma 5, for the u-unique $\hat{hc}_0 = \hat{\theta}(\hat{hc})$, while merging the tree functions via $\sqcup_{\mathcal{T}}$. It is important to note that $\hat{hc}_0 \neq \hat{hc}$, so the "tree" $\hat{\theta}'$ will not be "disconnected" after reattaching $\hat{hc}$ to the new abstract ancestor.

□

B Uniqueness Domain as Tracking the Cardinality of Abstraction

There is a trivial Galois connection between the counting and uniqueness domains:

$$\langle \wp(\mathbb{N}), \subseteq \rangle \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \langle \mathscr{U}, \sqsubseteq \rangle, \text{ where}$$

$$\alpha'(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ 1 & \text{if } S = \{1\} \\ + & \text{otherwise} \end{cases} \qquad \gamma'(u) = \begin{cases} \emptyset & \text{if } u = \bot \\ \{1\} & \text{if } u = 1 \\ \mathbb{N} & \text{otherwise} \end{cases}$$

Indeed, both $\langle \wp(\mathbb{N}), \subseteq \rangle$ and $\langle \mathscr{U}, \sqsubseteq \rangle$ are complete lattices, so for an arbitrary set A one can consider complete lattices of total functions $\langle A \to \wp(\mathbb{N}), \dot{\subseteq} \rangle$ and $\langle A \to \mathscr{U}, \dot{\sqsubseteq} \rangle$. There is a point-wise Galois connection between these lattices. The domain of counting functions $\langle A \to \wp(\mathbb{N}), \dot{\subseteq} \rangle$ answers the question *"for any element $a \in A$, how many times a has been used?"* and plays a role of a cardinality counter, which can be attached to each state of a computation to count, for instance, how many object in the particular context have been allocated. In the meanwhile, the domain of uniqueness functions $\mathcal{U}(A) = \langle A \to \mathscr{U}, \dot{\sqsubseteq} \rangle$ helps to answer the question *"whether some $a \in A$ has been used zero, one or more than one times?"* and works as an abstract counter, which, for instance, helps to know precisely, whether zero, one or more objects have been allocated in the particular context.

C Full Concrete and Abstract Semantics of A-Normal Featherweight Java with Ownership

In this section we give the full description of concrete semantics of A-Normal Featherweight Java with Ownership as well as well as of its abstract counterpart.

C.1 Concrete semantics of FJO

Concrete semantics is a binary relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$ in the form of a small-step CEK machine with heap-allocated continuations and an extra state component to maintain an ownership tree. The opaque functions $\mathcal{M}$ and $\mathcal{F}$ are standard lookup functions for class methods and fields respectively. The semantics is parametrized by two functions: *merge* and *record*, which serve for allocating local and heap contexts respectively. An interested reader should take a look on the work of Smaragdakis et al. on context-sensitive points-to analyses [24] for more details.

Variable reference

$$([\![v = v';^{\ell}]\!], B, \sigma, a_k, c, \theta) \Rightarrow (succ(\ell), B, \sigma', a_k, c, \theta), \text{ where}$$
$$\sigma' = \sigma + [B(v) \mapsto \sigma(B(v'))].$$

Return

$$([\![\texttt{return } v;^{\ell}]\!], B, \sigma, a_k, c, \theta) \Rightarrow (s, B', \sigma', a_k', c', \theta), \text{ where}$$
$$(a_{ret}, B', s, c', a_k') = \sigma(a_k) \qquad \sigma' = \sigma + [a_{ret} \mapsto \sigma(B(v))].$$

Field lookup

$$(\llbracket v = v'.f;^{\ell} \rrbracket, B, \sigma, a_k, c, \theta) \Rightarrow (succ(\ell), B, \sigma', a_k, c, \theta), \text{ where}$$
$$(_, [f \mapsto a_f]) = \sigma(B(v')) \qquad \sigma' = \sigma + [B(v) \mapsto \sigma(a_f)].$$

Field update

$$(\llbracket v.f = v';^{\ell} \rrbracket, B, \sigma, a_k, c, \theta) \Rightarrow (succ(\ell), B, \sigma', a_k, c, \theta), \text{ where}$$
$$(hc, _) = \sigma(B(v)) \qquad a_f = (f, hc) \qquad \sigma' = \sigma + [a_f \mapsto \sigma(B(v'))].$$

Method call

$$(\llbracket v = v_0.m(v');^{\ell} \rrbracket, B, \sigma, a_k, c, \theta) \Rightarrow (s_0, B', \sigma', a'_k, c', \theta), \text{ where}$$
$$M = \llbracket t\ m(t\ v'')\ \{\overrightarrow{t\ v'''}\ ;\ \overrightarrow{s}\} \rrbracket = \mathcal{M}(o_0, m)$$
$$o_0 = (hc_0, _) = \sigma(B(v_0)) \qquad o' = \sigma(B(v')) \qquad c' = merge(\ell, hc_0, c)$$
$$a' = (v'', c') \qquad a''_j = (v'''_j, c') \qquad B' = ([v'' \mapsto a', v'''_j \mapsto a''_j], hc_0)$$
$$k = (B(v), B, succ(\ell), c, a_k) \qquad a'_k = (m, c) \qquad \sigma' = \sigma + [a' \mapsto o'] + [a'_k \mapsto k]$$

Object allocation

$$(\llbracket v = \texttt{new}\ C\langle p \rangle;^{\ell} \rrbracket, B, \sigma, a_k, c, \theta) \Rightarrow (succ(\ell), B, \sigma', a_k, c, \theta'), \text{ where}$$
$$hc = record(\ell, c) \qquad \overrightarrow{f} = \mathcal{F}(C) \qquad a_i = (f_i, hc)$$
$$o' = (hc, [f_i \mapsto a_i]) \qquad \sigma' = \sigma + [B(v) \mapsto o'] \qquad \theta' = adjust(\theta, p, hc, B, \sigma)$$

C.2 Abstract semantics of FJO

Abstract semantics $(\rightsquigarrow) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ is a sound approximation of the abstract transition function. It is described in the form of a non-deterministic small-step abstract machine. The operation $\sqcup$ for stores merges the sets for the same key value. The rules comprising $\in$ when performing lokup into an abstract store $\hat{\sigma}$ are the source of the non-determinism.

Variable reference

$$(\llbracket v = v';^{\ell} \rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \rightsquigarrow (succ(\ell), \hat{B}, \hat{\sigma}', \hat{a}_k, \hat{c}, \hat{\theta}_u), \text{ where}$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{B}(v) \mapsto \hat{\sigma}(\hat{B}(v'))].$$

Return

$$(\llbracket \texttt{return}\ v;^{\ell} \rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \rightsquigarrow (s, \hat{B}', \hat{\sigma}', \hat{a}'_k, \hat{c}', \hat{\theta}_u), \text{ where}$$
$$(\hat{a}_{ret}, \hat{B}', s, \hat{c}', \hat{a}'_k) \in \sigma(\hat{a}_k) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_{ret} \mapsto \hat{\sigma}(\hat{B}(v))].$$

Field lookup

$$(\llbracket v = v'.f;^{\ell} \rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \rightsquigarrow (succ(\ell), \hat{B}, \hat{\sigma}', \hat{a}_k, \hat{c}, \hat{\theta}_u), \text{ where}$$
$$(_, [f \mapsto \hat{a}_f]) \in \hat{\sigma}(\hat{B}(v')) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{B}(v) \mapsto \hat{\sigma}(\hat{a}_f)].$$

Field update

$$(\llbracket v.f = v';^{\ell} \rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \rightsquigarrow (succ(\ell), \hat{B}, \hat{\sigma}', \hat{a}_k, \hat{c}, \hat{\theta}_u), \text{ where}$$
$$(\hat{hc}, _) \in \hat{\sigma}(\hat{B}(v)) \qquad \hat{a}_f = (f, \hat{hc}) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_f \mapsto \hat{\sigma}(\hat{B}(v'))].$$

Method call

$$(\llbracket v = v_0.m(v');^{\ell}\rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \rightsquigarrow (s_0, \hat{B}', \hat{\sigma}', \hat{a}'_k, \hat{c}', \hat{\theta}_u), \text{ where}$$

$$M = \llbracket t\ m(t\ v'')\ \{\overrightarrow{t\ v'''}\ ;\ \overrightarrow{s}\}\rrbracket = \mathcal{M}(\hat{o}_0, m)$$

$$\hat{o}_0 = (\hat{hc}_0, _) \in \hat{\sigma}(\hat{B}(v_0)) \qquad \hat{o}' \in \hat{\sigma}(\hat{B}(v')) \qquad \hat{c}' = \widehat{merge}(\ell, \hat{hc}_0, \hat{c})$$

$$\hat{a}' = (v'', \hat{c}') \qquad \hat{a}''_j = (v'''_j, \hat{c}') \qquad \hat{B}' = ([v'' \mapsto \hat{a}', v'''_j \mapsto \hat{a}''_j], \hat{hc}_0)$$

$$\hat{k} = (\hat{B}(v), \hat{B}, succ(\ell), \hat{c}, \hat{a}_k) \qquad \hat{a}'_k = (m, \hat{c}) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{o}'] \sqcup [\hat{a}'_k \mapsto \hat{k}]$$

Object allocation

$$(\llbracket v = \texttt{new}\ C\langle p\rangle;^{\ell}\rrbracket, \hat{B}, \hat{\sigma}, \hat{a}_k, \hat{c}, \hat{\theta}_u) \rightsquigarrow (succ(\ell), \hat{B}, \hat{\sigma}', \hat{a}_k, \hat{c}, \hat{\theta}'_{u'}), \text{ where}$$

$$\hat{hc} = \widehat{record}(\ell, \hat{c}) \qquad \overrightarrow{f} = \mathcal{F}(C) \qquad \hat{a}_i = (f_i, \hat{hc})$$

$$\hat{o}' = (\hat{hc}, [f_i \mapsto \hat{a}_i]) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{B}(v) \mapsto \hat{o}'] \qquad \hat{\theta}'_{u'} = \widehat{adjust}(\hat{\theta}_u, p, \hat{hc}, \hat{B}, \hat{\sigma})$$