Owners as Ombudsmen [*]

Multiple Aggregate Entry Points for Ownership Types

Johan Östlund   Tobias Wrigstad

Uppsala University

{first.last}@it.uu.se

Abstract

Traditional deep ownership types gives a strong notion of aggregate by enforcing the so-called *owners-as-dominators* property: every path from a system root to an object must pass through its owner. Consequently, encapsulated aggregates must have a single *bridge object* that mediates all external interaction with its internal objects.

In this paper, we argue for a novel model of ownership that relaxes the single bridge object constraint of traditional ownership types and allows several bridge objects to collectively define an aggregate with a shared representation. We call such bridge objects *ombudsmen* to emphasise their benevolent nature; all ombudsmen are created internal to the aggregate, purposely.

The resulting system brings the aggregate notion close to the component notion found in e.g., UML, and further allows expressing common programming patterns such as iterators without resorting to systems that give unclear guarantees, or require additional complex machinery such as read-only references.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Object-Oriented Programming, Aliasing, Confinement

1. Introduction

In classic, Clarkean ownership types [7] aggregates fully encapsulate their representation; the only way for an external object to interact with an aggregate's representation is through public methods on the aggregate's single bridge object. This is called the *owners-as-dominators property* and was first formulated by Clarke et al. [11].

The owners-as-dominators property of traditional ownership types is very restrictive and, as previously identified [3, 5, 20, 21] prevents common programming patterns that require multiple entry points into a single representation. The canonical example of such a pattern is an iterator to a linked list. Owners-as-dominators only allows direct references to a list's links from an iterator that cannot escape the list itself, which renders it pretty useless in practise.

In ownership types, each object introduces a new *context*, a nested subheap, usually called **`this`**, which holds its representation objects. Every object acts as the single entry point into its **`this`** context from the outside and no facilities exist that allow two objects to share a common representation. The leftmost diagram in Figure 1 shows an abstract picture of owners-as-dominators where contexts are depicted as boxes with a singe entry point object depicted as a filled circle. Arrows denote references, and red arrows are not allowed and programs which give rise to such references are rejected at compile-time. In terms of ownership diagrams, owners-as-dominators guarantee that arrows cannot cross the boundary of a context going inward. Consequently, d may not reference f, but e may reference d.

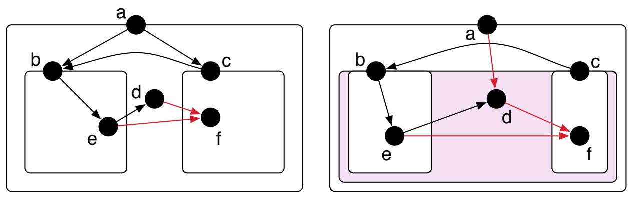

Figure 1. Owners-as-dominators (left) and owners-as-ombudsmen (right). The pink area denotes the shared aggregate context.

Contributions This paper contributes to the field of ownership types by proposing a static ownership types system in which each object gives rise to two contexts: *rep* and *aggregate*. The *rep* context is the equivalent to the **`this`** context of classical ownership systems and is defined solely by the object. The *aggregate* context, on the other hand, is novel and denotes a set of objects potentially shared between two or more objects *at the same level or nesting*, who define the context together. The resulting ownership structure is depicted to the right in Figure 1. The objects b and c now define a common context (coloured pink), that is protected from external access. We call b and c *ombudsmen* of their collaboratively defined aggregate, and—as the graph depicts—each act as a bridge object into *two* different, disjoint contexts.

Our design allows several objects to collaboratively construct an aggregate object and define multiple interaction points for it. We deliberately choose to limit objects to be part of a single aggregate, as we believe this is what a programmer expects, *i.e.,* it is "natural". Allowing multiple shared contexts is technically possible and a straightforward extension of our system.

As a consequence of our ombudsman design, an additional number of programming idioms can be expressed in ownership types systems, without compromising encapsulation. Owners-as-ombudsmen has a clear encapsulation property, which should be easy to understand and almost as powerful as owners-as-dominators.

Outline Section 2 describes the ombudsman concept in detail, states the properties of the system informally and how to type it, together with a couple of motivating examples. We then formally describe the system in Section 3 together with the key theorems. Section 4 discusses related work, Section 5 discusses encapsulation invariants and feature interaction, and Section 6 concludes.

[*] Supported in part by the Swedish Research Council within the UPMARC Linnaeus center of Excellence.

2. Ombudsmen

Our proposal allows several objects—*ombudsmen*—to act as bridge objects for a common aggregate. In terms of ownership types they share a common context. Objects in this context are "ombudsman-dominated", meaning that

> *every path from a root in the system to an object in an ombudsman-dominated context contains one of the context's ombudsmen.*

In the example in Figure 1, b and c are ombudsmen for a collaboratively defined aggregate. Their common aggregate context is depicted as the pink area. In addition, b and c each have a private context. The objects b and c, as well as objects in their private contexts, can reference objects in the aggregate context (*e.g.,* e refers d). Objects outside b or c may not refer to objects in their aggregate context (*e.g.,* a may not reference d). Objects in the aggregate context cannot reference the private contexts of their ombudsmen (*e.g.,* d may not refer to f).

As the leftmost picture of Figure 1 shows, in owners-as-dominators systems, every context is nested inside some other context, and references cannot cross a context from the outside to the inside. With owners-as-ombudsmen, contexts *on the same level of nesting* can *share a common context*, and references may cross from the "private contexts" of these objects into their shared context.

2.1 Typing Ombudsmen

The type system that lets us express aggregate encapsulation with multiple entry points is a straightforward extension of classic ownership types systems as found in *e.g.,* Joe$_1$ [8], Joline [9, 28] or OGJ [25]. Classes are parametrised over permissions to access external contexts and types instantiate those parameters with actual permissions, so-called *owner parameters*. From now on, we will use the word owner to denote a symbol in the program text that denotes a run-time context.

The first owner parameter of a type is the owner of the instance, available internally inside each class through the **owner** keyword. Additionally, each class knows the owners **rep**, **aggregate**, and **bridge**. The **rep** keyword denotes the private representation of the object and is equivalent to **this** in traditional ownership systems; **aggregate** denotes the (possibly) shared aggregate context; and finally **bridge** denotes a bridge object of the same aggregate as the current instance. Notably, if we think of an owner α as denoting the set of objects owned by α, then **bridge** $\subseteq$ **owner**.

In terms of the rightmost picture in Figure 1, the reference to b in c may have the owner **owner** or **bridge**; e's reference to d must have the owner **aggregate** (from the view of b, inside e it is some other owner parameter which will be bound to **aggregate** in b); and b's reference to e must have the owner **rep**.

Whether c refers to b as **owner** or **bridge** makes an important difference. Only in the second case can c know that b belongs to the same aggregate. In the first case, c simply would not know whether b was an ombudsman for the same aggregate, or not. Consequently, d can only be obtained from b if b has owner **bridge**. Otherwise, the path could denote the representation of a different aggregate, and would therefore not be safe to access.

A Small Example Figure 2 shows a minimal code example for expressing a component with two provided services `portOne` and `portTwo`, and a required service `required`. To a client, the objects in `portOne` and `portTwo` are siblings to the component—they have the same owner. Trying to obtain a reference to `portTwo` as an ombudsman is not possible, since the component's aggregate and the client's (if any) are not the same. Last, writing to a field containing an ombudsman is not possible externally, since external objects cannot tell what objects are ombudsmen for the same aggregate.

Discussion Notably, the **bridge** owner is only used by ombudsmen to identify other ombudsmen of the same aggregate. Any reference to an aggregate that is not internal to the aggregate or any of its ombudsmen is an ombudsman by definition, and there is no need to capture this in the type.

Well-formed construction of aggregate objects is one of the key considerations of our system design. Any ombudsman has the capability of constructing other ombudsmen and access the parts of their interface that mention **aggregate**. All ombudsmen are owned by **owner** (or its more specific subset context **bridge**), and consequently—all dominating objects are *siblings*. Coalescing existing objects created outside an aggregate with an aggregate is possible using ownership transfer [9, 22, 28]. In this case, one object must act as the "initial object" and move the unique objects into **bridge**. Such a method is easy to write and must be manually specified to prevent external objects from creating ad-hoc ombudsmen (see further Section 2.4) and thus get access to an aggregate's internals.

Although we have defined ombudsmen for the Joe/Joline family of ownership systems [8–10, 24], we believe they could easily be added to universe types [13, 21, 22] as well as OGJ [25], and similar.

We now continue our introduction to ombudsmen by way of a few example including two common programming idioms: components and external iterators.

2.2 Motivating Example: Components

Standard UML components are implemented as aggregates of collaborating objects [4]. A component may provide several different interfaces (aka required and provided services); different applications may use different interfaces or a combinations of different interfaces.

```
class Component<owner> {
  Data<aggregate> d = new Data<aggregate>;
  IService<bridge> portOne = new ServiceX<bridge>(d);
  IService<bridge> portTwo = new ServiceY<bridge>(d);
  IServiceC<owner> required;
}

class Client<owner> {
  Component<rep> c;
  IService<rep> generator = ... ;

  ...
  IService<rep> s1 = c.portOne;
  c.required = generator;
  IService<bridge> s2 = c.portOne; // Fails!!
  c.portOne = c.portOne; // Fails!!
}
```

Figure 2. Defining a component with two provided services and one required service.

With classic ownership types, a component that wishes to export several different interfaces must do so through a single object if encapsulation is to be retained. If each interface was implemented as a separate object, the objects would not be able to share any data unless that data could also be exposed outside the component, which is likely to destroy verification properties. To implement components with proper encapsulation, several objects must be able to share a common representation that cannot leak.

2.3 Motivating Example: Iterators with Ombudsmen

Figure 3 shows the ownership diagram for a linked list aggregate and Figure 4 the source code. Modulo the use of the novel **bridge** and **aggregate** owners, the code should be straightforward.

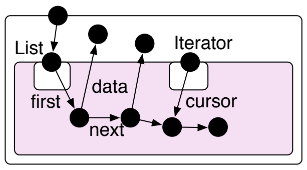

Figure 3. Encoding Iterators Using an Ombudsman.

In the example, the list's links are part of the aggregate, and the list has no private data. Initially, the list object is the only ombudsman through which the links can be manipulated. The `iterator` method in the list class creates and returns an ombudsman in the form of an iterator. As an ombudsman for the list aggregate, the iterator may reference the list's links in the `cursor` field. Any number of iterators may exist; the pattern would even allow several list objects that shared a common set of links—for whatever purpose.

Note that even though the `first` field in the list class is public, it will only be accessible to objects inside the aggregate, including other ombudsmen for the same aggregate. Thus, the list's links are properly encapsulated; they cannot be modified outside the list aggregate. However, unlike classic ownership types, the list aggregate has multiple ombudsmen, allowing access to the links from several disjoint entry points.

2.4 Attaching Ombudsmen and Incremental Aggregate Construction

Figure 5 shows an example of how external objects can be made part of an existing aggregate. We call this *attaching an ombudsman*, and makes use of ownership transfer in a straightforwardly fashion. In the example, a unique iterator object is passed to the iterator method, which is subsequently attached to the current aggregate by moving it into the **bridge** context. Following the syntax of Joline [9, 28], we use `--` to denote destructively reading the contents of a unique variable.

In a system using external uniqueness, such as Joline, any objects in the aggregate context of the iterator will be moved into the list aggregate. Thus, in systems supporting external uniqueness, aggregates can be constructed incrementally by attaching ombudsmen to each other, merging their aggregate contexts. This practise is sound as the uniquely referenced ombudsman is always a dominating node to any object inside its aggregate context, akin to the relation between an owner and its **this** context in classical ownership types.

```
class List<owner,data outside owner> {
  public Link<aggregate,data> first;

  Iterator<bridge,data> iterator() {
    Iterator<bridge,data> iter =
                        new Iterator<bridge,data>();
    iter.cursor = first;
    return iter;
  }
}

class Iterator<owner,data outside owner> {
  Link<aggregate,data> cursor;

  Object<data> next() {
    Object<data> value = cursor.data;
    cursor = cursor.next;
    return value;
  }
}

class Link<owner,data> {
  Link<owner,data> next;
  Object<data> data;
}
```

Figure 4. A list aggregate accessible via the list role and an iterator role. In the formalism, we have elided explicit nesting relations such as **outside** above.

```
Iterator<bridge,data> iterator(Iterator<unique,data> i) {
  Iterator<bridge,data> iter = i--; // move into bridge
  iter.cursor = first;
  return iter;
}
```

Figure 5. Coalescing an external iterator into the list aggregate.

3. Formalising Ombudsmen

Formalising ombudsmen is quite simple and the most relevant changes are in the type rules (EXPR-SELECT), (EXPR-UPDATE) and (EXPR-METHOD-CALL). Our formalism is inspired by [8, 28].

When reading or updating a field f of a non-bridge receiver x, f may not point to objects in x's aggregate. This is a straightforward adaptation of the static visibility constraint of Clarkean systems that reads $\textbf{rep} \in \mathsf{Owners}(\tau) \Rightarrow e = \textbf{this}$, which our system also uses. Further, if f under the same circumstances points to an ombudsman of x's aggregate, its type is reported to us as a sibling of x. This is visible in (EXPR-SELECT), (EXPR-UPDATE).

We consider only unary methods for simplicity and without loss of generality. Ombudsman adaptation is employed to translate internal to external types and there is an additional visibility constraint that prevents calling methods that expect ombudsman arguments, unless the receiver object is itself an ombudsman. The same constraint must hold for field update. This is visible in (EXPR-UPDATE) and (EXPR-METHOD-CALL).

Any type owned by **bridge** can be subsumed by the equivalent type owned by **owner**, since for all contexts, **bridge** denotes a subset of **owner**. This can be expressed trivially by adding an additional subtyping rule, see (BRIDGE-OWNER-SUBSUMPTION).

Conventions and Conveniences We follow the practise of FJ [17] and use an overbar notation for sequences of terms in the standard fashion. For example, $\overline{p}$ denotes a sequence $p_1, \ldots, p_n$ and $\overline{f : \tau}$ denotes a sequence $f_1 : \tau_1, \ldots f_n : \tau_n$ for $n \geq 0$. To turn such a sequence into a set, we write it within $\{\ \}$, *e.g.*, $\{\overline{p}\} = \{p \mid p \in \overline{p}\}$.

Like many ownership types papers before us [7, 9, 11, 24, 28], we sometimes write $\mathsf{C}\langle\sigma\rangle$ for a type $\mathsf{C}\langle\overline{p}\rangle$ where σ is a map from the names of the formal parameters of C to the actual parameters $\overline{p}$. For example, if C is declared $\textbf{class}\ \mathsf{C}\langle\textbf{owner}, \mathsf{a}, \mathsf{b}\rangle \cdots$ in the program, then if $\mathsf{C}\langle p_1, p_2, p_3\rangle$ is a well-formed type, we sometimes write $\mathsf{C}\langle\sigma\rangle$ for the implicitly defined $\sigma = \{\textbf{owner} \mapsto p_1, \mathsf{a} \mapsto p_2, \mathsf{b} \mapsto p_3\}$. As a further convenience, we sometimes write σ^p to mean $\sigma \cup \{\textbf{owner} \mapsto p\}$ and σ_p to mean $\sigma \cup \{\textbf{aggregate} \mapsto p\}$ (used in the dynamic semantics, possibly combined with σ^p).

$\vdash P : \tau$	P is a well-formed program with type τ
$\vdash C$	C is a well-formed class
$\vdash E, x : \tau$	E is extended by a variable x with type τ
$\vdash E, p\ \mathcal{R}\ q$	E is extended by a good nesting relation ($\mathcal{R} \in \{\prec^*, \succ^*\}$) between contexts p and q
$E; \tau \vdash F$	F is a well-typed field declaration and does not override a field in a supertype
$E; \tau \vdash M$	M is a well-typed method declaration overriding preserves typing
$E \vdash e : \tau$	e is a well-formed expression with type τ
$E \vdash p$	p is a good owner in the scope E
$E \vdash p\, \mathcal{R}\, q$	p is inside/outside q in the scope E; $\mathcal{R} \in \{\prec^*, \succ^*\}$
$E \vdash p \rightarrow^{\mathsf{ok}} q$	p may reference q in the scope E
$E \vdash \tau$	τ is a well-formed type in the scope E
$E \vdash \tau <: \tau'$	τ is a subtype of τ' in the scope E

Table 1. Judgements in the static system.

3.1 Static Semantics

Syntax The syntax of our system is as follows:

$$\begin{array}{lcll}
P & ::= & \overline{C}\quad \textbf{class}\ \mathtt{Object}\langle\textbf{owner}\rangle\ \{\ \}\quad e & (\textit{Program}) \\
C & ::= & \textbf{class}\ \mathtt{C}\langle\textbf{owner}, \overline{p}\rangle\ \textbf{extends}\ \mathtt{D}\langle\overline{p}\rangle\ \{\ \overline{F}\ \overline{M}\ \} & (\textit{Class decl.}) \\
F & ::= & \tau\ f & (\textit{Field decl.}) \\
M & ::= & \tau\ m(\tau\ x)\ \{\ e\ \} & (\textit{Method decl.}) \\
e & ::= & \textbf{let}\ x = e\ \textbf{in}\ e \mid x \mid x.f \mid x.f = y \mid & (\textit{Expressions}) \\
 & & x.m(y) \mid \textbf{null} \mid \textbf{new}\ \tau & \\
\tau & ::= & \mathtt{C}\langle\overline{p}\rangle \mid \mathtt{C}\langle\overline{\kappa}\rangle & (\textit{Types})
\end{array}$$

where the grey box is an extensions in the dynamic semantics for run-time types. Metavariables x, y are used for names of variables (including **this**), p, q for names of contexts (including **rep**, **owner**, **bridge** and **aggregate**). For simplicity, local variables and sequences are encoded using standard let-expressions.

For the static semantics, we use a type environment E of the standard fashion, containing mappings from local variables to types and relations between contexts in the current scope: $E ::= \epsilon \mid E, x : \tau \mid E, p \prec^* q \mid E, p \succ^* q$. Declarations and let-expressions extend E in a straightforward fashion. Table 1 shows an overview of the judgements used in our static system.

Class Table and Helper Predicates $\mathsf{OmbudsmanAdaptation}$, a key helper predicate, is defined thus:

$$\begin{array}{rcl}
\mathsf{OmbudsmanAdaptation}(\textbf{bridge}, \tau) & = & \tau \\
\mathsf{OmbudsmanAdaptation}(p, \tau) & = & \tau\{^{\textbf{owner}}/_{\textbf{bridge}}\}
\end{array}$$

where $p \neq$ **bridge** is assumed in the last case. This predicate is used to change the internal view of an object as a bridge object to the *current* aggregate to the external view of an object—a bridge object for *some* aggregate at the same nesting depth.

For every class $\mathtt{C}$, we assume the existence of a class table $\mathcal{CT}(\mathtt{C})$, such that if $\textbf{class}\ \mathtt{C}\langle\overline{p}\rangle\ \textbf{extends}\ \mathtt{D}\langle\sigma\rangle\ \{\ F\ M\ \}$, $\mathcal{CT}(\mathtt{C}) = (F, M) \circ \sigma(\mathcal{CT}(\mathtt{D}))$. We define $\mathcal{CT}(\mathtt{Object}) = (\emptyset, \emptyset)$. Further, we define lookup in the class table such that $\mathcal{CT}(\mathtt{C})(f) = F(f)$ when $F(f) \neq \bot$, else $\mathcal{CT}(\mathtt{C})(f) = \sigma(\mathcal{CT}(\mathtt{D})(f))$, and similar for looking up m. $F(f) = \tau$ if $\tau f \in F$, else $\bot$. Isomorphically, $M(m) = (\tau_1 \to \tau_2, x, e)$ if $\tau_2\ m(\tau_1\ x)\ \{\ e\ \} \in M$, else $\bot$.

Given the class table, we can define lookup helper predicates in a straightforward fashion.

$$\begin{array}{rcl}
\mathsf{FieldType}(\mathtt{C}, f) & = & \mathcal{CT}(\mathtt{C})(f) \\
\mathsf{Signature}(\mathtt{C}, m) & = & \mathit{fst}(\mathcal{CT}(\mathtt{C})(m)) \\
\mathsf{Param}(\mathtt{C}, m) & = & \mathit{snd}(\mathcal{CT}(\mathtt{C})(m)) \\
\mathsf{Body}(\mathtt{C}, m) & = & \mathit{trd}(\mathcal{CT}(\mathtt{C})(m)) \\
\mathsf{Fields}(\mathtt{C}) & = & \{f \mid \mathcal{CT}(\mathtt{C})(f) \neq \bot\} \\
\mathsf{Owners}(\mathtt{C}\langle\overline{p}\rangle) & = & \{\overline{p}\} \\
\sigma(\mathtt{C}\langle\overline{p}\rangle) & = & \mathtt{C}\langle\sigma(\overline{p})\rangle
\end{array}$$

We assume the existence of a predicate $\mathsf{Arity}(\mathtt{C})$ that returns the number of owner parameters, including **owner**, declared for the class $\mathtt{C}$, *e.g.,* $\mathsf{Arity}(\mathtt{List}) = 2$ from the example in Figure 4.

Declarations A program is well-formed if all its classes are well-formed and the starting expression of the program is well-typed. For simplicity, the root class `Object` is treated special.

$$\text{(WF-PROGRAM)}\quad \frac{\vdash \overline{C} \qquad \epsilon \vdash e : \tau}{\vdash \overline{C}\quad \textbf{class}\ \mathtt{Object}\langle\textbf{owner}\rangle\ \{\ \}\quad e : \tau}$$

A class is well-formed if its fields and methods are well-formed, the owner parameters passed to the super class are good (respect the nesting), and **owner** is only used in the first position of the owner formals.

$$\text{(WF-CLASS)}\quad \frac{\begin{array}{c}E = \textbf{owner} \prec^* \textbf{world}, \textbf{rep} \prec^* \textbf{owner}, \textbf{bridge} \prec^* \textbf{owner}, \backslash \\ \textbf{aggregate} \prec^* \textbf{owner}, \overline{p} \succ^* \textbf{owner}, \textbf{this} : \mathtt{C}\langle\textbf{bridge}, \overline{p}\rangle \\ \{\overline{q}\} \subseteq \{\overline{p}\} \qquad \textbf{owner} \notin \{\overline{p}\} \\ \tau_{\mathsf{s}} = \mathtt{D}\langle\textbf{owner}, \overline{q}\rangle \qquad E \vdash \tau_{\mathsf{s}} \qquad E; \tau_{\mathsf{s}} \vdash \overline{F} \qquad E; \tau_{\mathsf{s}} \vdash \overline{M}\end{array}}{\vdash \textbf{class}\ \mathtt{C}\langle\textbf{owner}, \overline{p}\rangle\ \textbf{extends}\ \mathtt{D}\langle\overline{q}\rangle\ \{\ \overline{F}\ \overline{M}\ \}}$$

A field is well-typed if its type is valid in the current scope, and there is no equi-named field in a superclass.

$$\text{(WF-FIELD)}\quad \frac{E \vdash \mathtt{C}\langle\sigma\rangle \qquad E \vdash \tau \qquad \mathsf{FieldType}(\mathtt{C}, f) = \bot}{E; \mathtt{C}\langle\sigma\rangle \vdash \tau\ f}$$

A method is well-formed if its types are well-formed in the current scope, its method body corresponds to the declared return type, and overriding preserves types.

$$\text{(WF-METHOD)}\quad \frac{\begin{array}{c}E \vdash \mathtt{C}\langle\sigma\rangle \qquad E \vdash \tau \qquad E, x : \tau' \vdash e : \tau \\ \mathsf{Signature}(\mathtt{C}, m) = \bot \vee \mathsf{Signature}(\mathtt{C}, m) = \sigma(\tau') \to \sigma(\tau)\end{array}}{E; \mathtt{C}\langle\sigma\rangle \vdash \tau\ m(\tau'\ x)\ \{\ e\ \}}$$

Expressions Each expression is typed given the static type information E derived initially for each method in (WF-CLASS), and extended with variables by (WF-METHOD) and (EXPR-LET). For simplicity, we assume that all variables have unique names.

$$\text{(EXPR-LET)}\quad \frac{E \vdash e' : \tau' \quad E, x : \tau' \vdash e : \tau}{E \vdash \textbf{let}\ x = e'\ \textbf{in}\ e : \tau} \qquad \text{(EXPR-VAR)}\quad \frac{\vdash E \qquad E(x) = \tau}{E \vdash x : \tau}$$

Reading a field of an object makes use of two key constraints: first, the two visibility constraints that say representation objects may only be accessed through the special **this** receiver, which is due to Clarke et al. [11] and that aggregate objects may only be accessed through ombudsmen. Last, we apply the $\mathsf{OmbudsmanAdaptation}$ helper function that make ombudsmen appear as regular objects when viewed externally.

$$\text{(EXPR-SELECT)}\quad \frac{\begin{array}{c}E \vdash x : \mathtt{C}\langle\sigma^p\rangle \\ \mathsf{FieldType}(\mathtt{C}, f) = \tau \\ \textbf{rep} \in \mathsf{Owners}(\tau) \Rightarrow x = \textbf{this} \\ \textbf{aggregate} \in \mathsf{Owners}(\tau) \Rightarrow p = \textbf{bridge} \\ \mathsf{OmbudsmanAdaptation}(p, \tau) = \tau'\end{array}}{E \vdash x.f : \sigma^p(\tau')}$$

(EXPR-UPDATE) is very similar to (EXPR-SELECT), although it does not use OmbudsmanAdaptation (that would not be sound as we are writing, not reading a field—similar to wildcards in Java generics) and places an additional restriction on (EXPR-UPDATE): a field holding an ombudsman can only be accessed through another ombudsman.

$$\text{(EXPR-UPDATE)}\quad \dfrac{\begin{array}{c} E \vdash x : \mathtt{C}\langle\sigma^p\rangle \\ \mathsf{FieldType}(\mathtt{C}, f) = \tau \\ E \vdash y : \sigma^p(\tau) \\ \mathbf{rep} \in \mathsf{Owners}(\tau) \Rightarrow x = \mathbf{this} \\ \mathbf{bridge}, \mathbf{aggregate} \in \mathsf{Owners}(\tau) \Rightarrow p = \mathbf{bridge} \end{array}}{E \vdash x.f = y : \sigma^p(\tau)}$$

The static semantics for calling a method is straightforward and contains the amalgamation of the restrictions of (EXPR-SELECT) and (EXPR-UPDATE) as well as uses OmbudsmanAdaptation so that returning a **bridge** object from an invocation on a non-bridge object type loses its "bridge status" (from the view of the type system).

$$\text{(EXPR-METHOD-CALL)}\quad \dfrac{\begin{array}{c} E \vdash x : \mathtt{C}\langle\sigma^p\rangle \\ \mathsf{Signature}(\mathtt{C}, m) = \tau_1 \to \tau_2 \\ E \vdash y : \sigma^p(\tau_1) \\ \mathbf{rep} \in \mathsf{Owners}(\tau_1) \cup \mathsf{Owners}(\tau_2) \Rightarrow x = \mathbf{this} \\ \mathbf{bridge}, \mathbf{aggregate} \in \mathsf{Owners}(\tau_1) \Rightarrow p = \mathbf{bridge} \\ \mathbf{aggregate} \in \mathsf{Owners}(\tau_2) \Rightarrow p = \mathbf{bridge} \\ \mathsf{OmbudsmanAdaptation}(p, \tau_2) = \tau \end{array}}{E \vdash x.m(y) : \sigma^p(\tau)}$$

The static semantics for **null** and instantiation are straightforward. Last, (EXPR-SUBSUMPTION) allows the type of an expression to be subsumed into a supertype.

$$\text{(EXPR-NULL)}\ \dfrac{E \vdash \tau}{E \vdash \mathbf{null} : \tau} \qquad \text{(EXPR-NEW)}\ \dfrac{E \vdash \tau}{E \vdash \mathbf{new}\ \tau : \tau} \qquad \text{(EXPR-SUBSUMPTION)}\ \dfrac{\begin{array}{c} E \vdash e : \tau' \\ E \vdash \tau' <: \tau \end{array}}{E \vdash e : \tau}$$

Type Environment Construction The static type environment E follows standard practises.

$$\text{(E-}\epsilon\text{)}\ \dfrac{}{\vdash \epsilon} \qquad \text{(E-VAR)}\ \dfrac{E \vdash \tau \qquad x \notin dom(E)}{\vdash E, x : \tau}$$

$$\text{(E-CONTEXT)}\ \dfrac{E \vdash q \qquad p \notin dom(E) \qquad \mathcal{R} \in \{\prec^*, \succ^*\}}{\vdash E, p\ \mathcal{R}\ q}$$

Contexts Statically, contexts are added to the environment in (WF-CLASS). The only manifest owner is (WORLD).

$$\text{(GOOD-CONTEXT)}\ \dfrac{\vdash E \qquad p \in dom(E)}{E \vdash p} \qquad \text{(GOOD-WORLD)}\ \dfrac{\vdash E}{E \vdash \mathbf{world}}$$

Rules for nesting relations are straightforward and follow a wealth of ownership papers in the Clarkean family.

$$\text{(INSIDE)}\ \dfrac{\vdash E \qquad p \prec^* q \in E}{E \vdash p \prec^* q} \qquad \text{(OUTSIDE)}\ \dfrac{\vdash E \qquad q \succ^* p \in E}{E \vdash p \prec^* q}$$

$$\text{(INSIDE-REFLEXIVE)}\ \dfrac{E \vdash p}{E \vdash p \prec^* p} \qquad \text{(INSIDE-TRANSITIVE)}\ \dfrac{E \vdash p \prec^* p' \qquad E \vdash p' \prec^* q}{E \vdash p \prec^* q}$$

Permissions Permissions in our system govern how references may cross context boundaries. Inside nesting implies permission to reference, just like in classical ownership types in (P-INSIDE).

$$\text{(P-INSIDE)}\ \dfrac{E \vdash p \prec^* q}{E \vdash p \to^{\mathsf{ok}} q} \qquad \text{(P-REP)}\ \dfrac{\vdash E \qquad p \in \{\mathbf{bridge}, \mathbf{aggregate}\}}{E \vdash \mathbf{rep} \to^{\mathsf{ok}} p}$$

Additionally, an ombudsman's private representation may reference its aggregate in (P-REP).

Types and Subtyping In our system, a type is well-formed if its owner has the right to reference all its owner parameters, and additionally, the number of parameters must correspond to the class declaration.

$$\text{(GOOD-TYPE)}\ \dfrac{E \vdash p \qquad E \vdash p \to^{\mathsf{ok}} \overline{p} \qquad \mathsf{Arity}(\mathtt{C}) = |p, \overline{p}|}{E \vdash \mathtt{C}\langle p, \overline{p}\rangle}$$

Subtyping follows the same rules as for classic ownership types. Reference permissions are propagated upward in the class hierarchy by the forwarding in the class declaration, and the subtyping relation is reflexive and transitive.

$$\text{(SUBTYPE-DIRECT)}\ \dfrac{E \vdash \mathtt{C}\langle\sigma\rangle \qquad \mathbf{class}\ \mathtt{C}\langle\cdots\rangle\ \mathbf{extends}\ \mathtt{D}\langle\overline{q}\rangle \cdots \in P}{E \vdash \mathtt{C}\langle\sigma^p\rangle <: \mathtt{D}\langle p, \sigma(\overline{q})\rangle}$$

$$\text{(SUBTYPE-TRANS)}\ \dfrac{E \vdash \tau_1 <: \tau_3 \qquad E \vdash \tau_3 <: \tau_2}{E \vdash \tau_1 <: \tau_2} \qquad \text{(SUBTYPE-REFLEXIVE)}\ \dfrac{E \vdash \tau}{E \vdash \tau <: \tau}$$

The single novel subtyping rule in our system allows an ombudsman to be subsumed by its owner. This is required to safely export an ombudsman outside of its (aggregate) representation without compromising safety.

$$\text{(BRIDGE-OWNER-SUBSUMPTION)}\ \dfrac{E \vdash \mathtt{C}\langle\mathbf{bridge}, \overline{p}\rangle \qquad E \vdash \mathtt{C}\langle\mathbf{owner}, \overline{p}\rangle}{E \vdash \mathtt{C}\langle\mathbf{bridge}, \overline{p}\rangle <: \mathtt{C}\langle\mathbf{owner}, \overline{p}\rangle}$$

As an example of the use of this practise, see the list iterator example. Internally, the list's view of its iterator is `Iterator`⟨**bridge**, `data`⟩, but when obtained from some external object, the iterator's type is `Iterator`⟨**owner**, `data`⟩. This is sound since **bridge** always denotes a subset of **owner**.

3.2 Dynamic Semantics

The dynamic semantics is a big-step operational semantics and should be straightforward to follow. Objects are represented by triples with a type compartment, aggregate context id α, and a field compartment. Run-time types are the same as static types, but static owner names are substituted for run-time contexts. Run-time contexts are κ, either an object identity, an aggregate context identifier α, or the special context **world**.

$$\begin{array}{llll} H & ::= & [\,] \mid H[\iota \mapsto (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, F)] & \textit{(Heap)} \\ B & ::= & \epsilon \mid B, x \mapsto v \mid B, p \mapsto \kappa & \textit{(Bindings)} \\ \mathcal{F} & ::= & [\,] \mid \mathcal{F}[f \mapsto v] & \textit{(Fields)} \\ v & ::= & \epsilon \mid \iota & \textit{(Values)} \\ \kappa & ::= & \iota \mid \alpha \mid \mathbf{world} & \textit{(Contexts)} \end{array}$$

A configuration is a triple $\langle H; B, e\rangle$ of a heap H, bindings of variables to values and context names to contexts B, and an expression e. The initial configuration is $\langle [\,]; \emptyset, e\rangle$ that is, an empty heap, empty bindings and the program's start expression.

Rules (D-LET) and (D-VAR) are unsurprising. (D-LET) evaluates the expression e and binds the value v' to the variable x in the environment under which e' is evaluated. (D-VAR) just looks up the value bound to x in the frame.

$$\overset{\text{(D-LET)}}{\dfrac{\langle H; B, e\rangle \to \langle H', v'\rangle \qquad \langle H'; B, x \mapsto v', e'\rangle \to \langle H'', v''\rangle}{\langle H; B, \mathbf{let}\ x = e\ \mathbf{in}\ e'\rangle \;\to\; \langle H'', v''\rangle}} \qquad \overset{\text{(D-VAR)}}{\dfrac{B(x) = v}{\langle H; B, x\rangle \;\to\; \langle H, v\rangle}}$$

Looking up a field on an object receiver is straightforward. We write $H(\iota.f)$ as a shorthand for $\mathcal{F}(f)$ when $H(\iota) = (\mathsf{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F})$.

$$\overset{\text{(D-SELECT)}}{\dfrac{B(x) = \iota \qquad H(\iota.f) = v}{\langle H; B, x.f\rangle \;\to\; \langle H, v\rangle}}$$

Updating a field on an object receiver is straightforward. We write $H(\iota.f) := v$ as a shorthand for $H[\iota \mapsto (\mathsf{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F}[f \mapsto v])]$ when $H(\iota) = (\mathsf{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F})$.

$$\overset{\text{(D-UPDATE)}}{\dfrac{B(x) = \iota \qquad B(y) = v}{\langle H; B, x.f = y\rangle \;\to\; \langle H(\iota.f) := v, \epsilon\rangle}}$$

In our simple semantics, method calls are simply inlined in the current expression, and all owner names are substituted for their run-time equivalences, which are derived from the current **this**. Furthermore, **this** is substituted for the current object, and the parameter is substituted for the actual argument value.

$$\overset{\text{(D-METHOD-CALL)}}{\dfrac{\begin{array}{c} B(x) = \iota \qquad B(y) = v \qquad H(\iota) = (\mathsf{C}\langle\sigma^{\kappa}\rangle, \alpha, \_) \\ \mathsf{Body}(\mathsf{C}, m) = e \qquad \mathsf{Param}(\mathsf{C}, m) = x \\ B' = \mathbf{rep} \mapsto \iota, \mathbf{bridge} \mapsto \kappa, \mathbf{this} \mapsto \iota, x \mapsto v, \mathbf{aggregate} \mapsto \alpha \\ \langle H; B', \sigma^{\kappa}, e\rangle \;\to\; \langle H', v'\rangle \end{array}}{\langle H; B, x.m(y)\rangle \;\to\; \langle H', v'\rangle}}$$

N.B., ϵ denotes the run-time representation of **null**.

$$\overset{\text{(D-NULL)}}{\dfrac{}{\langle H; B, \mathbf{null}\rangle \;\to\; \langle H, \epsilon\rangle}}$$

Object creation is simple due to the absence of constructors and custom field initialisation. A fresh object has all its field initialised to **null** and a fresh context α is picked to represent its aggregate, unless it is a ombudsman, in which case the aggregate context is that of the current object.

$$\overset{\text{(D-NEW)}}{\dfrac{\begin{array}{c} \mathcal{F} = [f \mapsto \epsilon \mid f \in \mathsf{Fields}(\mathsf{C})] \qquad \iota \text{ is fresh} \\ p \neq \mathbf{bridge} \Rightarrow \alpha \text{ is fresh} \quad p = \mathbf{bridge} \Rightarrow \alpha = B(\mathbf{aggregate}) \end{array}}{\langle H; B, \mathbf{new}\ \mathsf{C}\langle p, \overline{p}\rangle\rangle \;\to\; \langle H[\iota \mapsto (\mathsf{C}\langle B(p), B(\overline{p})\rangle, \alpha, \mathcal{F})], \iota\rangle}}$$

For brevity, we omit the straightforward error trapping rules for dereferencing null pointers and propagating errors.

3.3 Meta Theory

In our reasoning about well-formedness, we rely on a combined type environment and store type $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, \iota : \tau \mid \Gamma, \alpha : \kappa \mid \Gamma, \mathbf{o}\,\iota : \alpha$. The entry $\alpha : \kappa$ maps an aggregate context α to the owner κ of all its ombudsmen. In a similar fashion, the entry $\mathbf{o}\,\iota : \alpha$ maps an object ι into an aggregate context α for which it acts as an ombudsman. Table 2 overviews the judgements in the meta theory.

$\vdash \Gamma$	Γ is a well-formed store type
$\Gamma \vdash \langle H; B, e\rangle : \tau$ $\Gamma \vdash \langle H; B, v\rangle : \tau$	$\langle H; B, e/v\rangle$ is a well-formed configuration with type τ under Γ
$\Gamma \vdash \mathsf{C}\langle\overline{\kappa}\rangle$	$\mathsf{C}\langle\overline{\kappa}\rangle$ is a well-formed type under Γ
$\Gamma \vdash \kappa \to^{\mathsf{ok}} \kappa'$	Objects in context κ have the permission to reference objects immediately in κ' under Γ
$\Gamma \vdash H$	H is a well-formed heap under Γ
$\Gamma \vdash v : \tau$	Value v has type τ under Γ

Table 2. Judgements in the meta-theoretic part of the formalism.

$$\overset{(\Gamma\text{-}\epsilon)}{\dfrac{}{\vdash \epsilon}} \qquad \overset{(\Gamma\text{-VAR})}{\dfrac{x \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}} \qquad \overset{(\Gamma\text{-OBJECT})}{\dfrac{\iota \notin \mathit{dom}(\Gamma) \qquad \Gamma \vdash \tau}{\vdash \Gamma, \iota : \tau}}$$

The rules (Γ-BRIDGE) and (Γ-AGGREGATE) are key elements in our system; in a well-formed store type, all ombudsmen of the same aggregate have the same owner.

$$\overset{(\Gamma\text{-BRIDGE})}{\dfrac{\begin{array}{c}\Gamma \vdash \kappa \\ \alpha \notin \mathit{dom}(\Gamma)\end{array}}{\vdash \Gamma, \alpha : \kappa}} \qquad \overset{(\Gamma\text{-AGGREGATE})}{\dfrac{\begin{array}{c}\vdash \Gamma \qquad \mathbf{o}\,\iota \notin \mathit{dom}(\Gamma) \\ \alpha : \kappa \in \Gamma \qquad \Gamma(\iota) = \mathsf{C}\langle\sigma^{\kappa}\rangle\end{array}}{\vdash \Gamma, \mathbf{o}\,\iota : \alpha}}$$

A well-formed heap can be extended by an object whose field contents correspond to that of the class declaration. All ombudsmen for the same aggregate must have the same owner.

$$\overset{\text{(HEAP-[])}}{\dfrac{\vdash \Gamma}{\Gamma \vdash [\,]}} \qquad \overset{\text{(HEAP-OBJECT)}}{\dfrac{\begin{array}{c}\Gamma(\iota) = \mathsf{C}\langle\sigma^{\kappa}\rangle \quad \Gamma(\mathbf{o}\,\iota) = \alpha \quad \Gamma(\alpha) = \kappa \quad \Gamma \vdash H \\ \Gamma \vdash \overline{v} : (\sigma^{\kappa}_{\alpha} \cup \{\mathbf{rep} \mapsto \iota, \mathbf{bridge} \mapsto \kappa\})(\overline{\tau}) \\ \mathsf{Fields}(\mathsf{C}) = \{\overline{f}\} \qquad \mathsf{FieldType}(\mathsf{C}, \overline{f}) = \overline{\tau}\end{array}}{\Gamma \vdash H, \iota \mapsto (\mathsf{C}\langle\sigma^{\kappa}\rangle, \alpha, [\overline{f} \mapsto \overline{v}])}}$$

A configuration is well-formed given an environment Γ if its heap is well-formed and its expression/value is well-typed.

$$\overset{\text{(GOOD-CONFIGURATION)}}{\dfrac{\Gamma \vdash H \qquad \Gamma \vdash e\,\{B\} : \tau\,\{B\}}{\Gamma \vdash \langle H; B, e\rangle : \tau}} \qquad \overset{\text{(GOOD-FINAL-CONFIGURATION)}}{\dfrac{\Gamma \vdash H \qquad \Gamma \vdash v : \tau\,\{B\}}{\Gamma \vdash \langle H; B, v\rangle : \tau}}$$

We assume a function $e/\tau\,\{B\}$ that replaces static names of owners in the domain of B with their dynamic counterparts, *e.g.,* $\mathsf{C}\langle p\rangle\,\{B\} = \mathsf{C}\langle B(p)\rangle$. The judgements $\Gamma \vdash e : \tau$ are copy and patch from the corresponding $E \vdash e : \tau$ and therefore omitted.

$$\overset{\text{(NULL-TYPE)}}{\dfrac{\Gamma \vdash \tau}{\Gamma \vdash \epsilon : \tau}} \qquad \overset{\text{(OBJECT-TYPE)}}{\dfrac{\Gamma(\iota) = \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \iota : \tau}}$$

Rules for good dynamic contexts are similar to their static counterparts.

$$\overset{\text{(CONTEXT-WORLD)}}{\dfrac{\vdash \Gamma}{\Gamma \vdash \mathbf{world}}} \qquad \overset{\text{(CONTEXT-OBJECT/AGGREGATE)}}{\dfrac{\vdash \Gamma \qquad \kappa \in \mathit{dom}(\Gamma)}{\Gamma \vdash \kappa}}$$

A type is well-formed if its owner has the right to reference the remaining owner parameters.

$$\overset{\text{(D-TYPE)}}{\dfrac{\Gamma \vdash \kappa \to^{\mathsf{ok}} \overline{\kappa} \qquad \mathsf{Arity}(\mathsf{C}) = |\kappa, \overline{\kappa}|}{\Gamma \vdash \mathsf{C}\langle\kappa, \overline{\kappa}\rangle}}$$

Modulo for aggregates and bridges, relations between contexts are not explicitly stored in Γ. Instead we infer them from the types present in a well-formed Γ. Reflexivity and transitivity of this relation are trivial and therefore omitted.

$$\frac{\vdash \Gamma \quad \Gamma(\kappa) = \mathsf{C}\langle\overline{\kappa}\rangle \quad \kappa' \in \overline{\kappa}}{\Gamma \vdash \kappa \prec^* \kappa'}\ \text{(D-INSIDE)} \qquad \frac{\vdash \Gamma \quad \Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \prec^* \kappa}\ \text{(D-OMBUDSMAN)}$$

Last, a context κ may reference another context κ' if an inside relation can be inferred from the first to the second, or if the second is an aggregate context and the first is inside an object defining it.

$$\frac{\Gamma \vdash \kappa \prec^* \kappa' \quad \vee \quad \Gamma \vdash \kappa \prec^* \iota \wedge \Gamma(\mathbf{o}\,\iota) = \kappa'}{\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \kappa'}\ \text{(D-MAYREF)}$$

We define $\rightarrow$ ("points to") and $\sqsupseteq$ ("aggregates") as binary relations between objects for some heap H such that $\iota \rightarrow \iota' \iff \exists\, f\ s.t.\ H(\iota.f) = \iota'$ and $\iota \sqsupseteq \iota' \iff H(\iota) = (\mathsf{C}\langle\overline{\kappa}\rangle, \alpha, \_) \wedge H(\iota') = (\mathsf{C}\langle\alpha, \_\rangle, \_, \_)$. We can now define "owners-as-ombudsmen" as a straightforward extension to owners-as-dominators.

Theorem 1: Owners as Ombudsmen For any two objects ι_1, ι_2 in a well-formed heap, $\iota_1 \rightarrow \iota_2 \Rightarrow \iota_1 \prec^* \mathsf{Owner}(\iota_2) \vee \exists\, \iota \sqsupseteq \iota_2 \wedge \iota_1 \prec^* \iota$.

In plain language this states that if an object ι_1 refers another object ι_2, then either ι_2 is a dominator of that object (the non-savvy ownership reader can consult *e.g.,* Clarke's dissertation [7] for additional details), or ι_2 is part of some aggregate and ι_1 is inside the private representation of this aggregate.

Proof (Sketch) Assume the existence of two well-formed objects ι_1, ι_2 such that $\iota_1 \rightarrow \iota_2$ and $\iota_1 \not\prec^* \mathsf{Owner}(\iota_2)$ and $\nexists\, \iota \sqsupseteq \iota_2 \wedge \iota_1 \prec^* \iota$. Clearly, for $\iota_1 \rightarrow \iota_2$, the class of ι_1 must be able to name the owner of ι_2. Statically, this owner must be either some class parameter p, **rep**, **aggregate**, **bridge** or **world**. The last case clearly must not be the case as world is outside anything. If the owner is p, **rep** or **owner**, then clearly $\iota_1 \prec^* \mathsf{Owner}(\iota_2)$ holds, since **rep** $= \iota_1$ and $\iota_1 \prec^*$ **owner** and $\iota_1 \prec^* p$ follows from (D-INSIDE). If the owner is **aggregate**, then by (HEAP-OBJECT), it must dynamically be an α s.t. $\Gamma(\alpha) = \kappa$ and $\mathbf{o}\,\iota_2 : \alpha \in \Gamma$. Clearly, this cannot be the case since then $\iota_1 \sqsupseteq \iota_2$ which violates the initial assumption. Now remains the case for **bridge**, which is similar to **owner** due to the subsumption **bridge** $\mapsto \kappa$ made in (HEAP-OBJECT). Thus, the proposition holds. □

Theorem 2: Subject Reduction We prove subject reduction in the standard fashion of progress plus preservation.

PROGRESS: If $\Gamma \vdash \langle H; B, e\rangle : \tau$, then $\langle H; B, e\rangle \rightarrow \langle H', v\rangle$ or $\langle H; B, e\rangle \rightarrow \mathsf{NullPointerException}$.

Proof The proof is straightforward by structural induction on the shape of e where most cases are immediate. The slightly more intricate cases, (D-SELECT), (D-UPDATE) and (D-METHOD-CALL) are all guarded by ($*$-NPE) versions (elided in this presentation) of the rule that handle null-dereferencing. By (HEAP-OBJECT), a well-formed object $(\tau, \alpha, \mathcal{F})$ has all its expected fields in $\mathcal{F}$, with the expected types, therefore, evaluation cannot get stuck accessing a non-existent field, and a similar argument applies to method calls. The only way evaluation can get stuck in our system is by infinite recursion, which is a mere technicality which is easy too handle by limiting stack space, see *e.g.,* [14, 28]. □

PRESERVATION: If $\Gamma \vdash \langle H; B, e\rangle : \tau$ and $\langle H; B, e\rangle \rightarrow^* \langle H', v\rangle$, then exists $\Gamma' \supseteq \Gamma$ s.t. $\Gamma' \vdash \langle H'; B, v\rangle : \tau$.

Although B might be updated by evaluating e, such updates will only be of local variables—not owners, which are the interesting elements of B w.r.t. final configurations.

Proof The proof is straightforward by structural induction on the shape of e. There are no surprising cases. □

4. Related Work

Several researchers have proposed extensions or deviations from traditional ownership types that overcome the single bridge object-problem. The encapsulation property of Universe Types [21], owners-as-modifiers, relaxes owners-as-dominators for read-only references. Thus, traditional universe types can express the iterator pattern, but only allow obtaining read-only references through list data via the iterators (modulo expensive and unsafe downcasts). Generic Universe Types [13] overcome this limitation, but do not allow iterators that change its originating collection. In summary, Universe types allow multiple entry points to aggregates, but only one of these entry points may have mutating capabilities. Furthermore, the multiple entry points can be exported arbitrarily in the system. Clarke et al. [10] similarly relax owner-as-dominators for *safe* references (that may only be used to read immutable parts of objects), and for references to immutable data.

In an ill-named paper, Boyapati et. al [5] allow relaxing owners-as-dominators for instances of inner classes. A list may define an inner iterator class that can be exported arbitrarily, but still access the enclosing object's representation. This allows expressing mutating and non-mutating iterators, but at the same time destroys the strong encapsulation, as there is no way for a type system to detect whether a backdoor to an object's representation exist or not.

Boyapati's proposal is somewhat close in spirit to ours: a single object starts as the initial bridge object for an aggregate, and may create additional bridge objects internally. However, a closer look reveals several shortcomings, which our system avoids:

Non-modular All bridge objects must be defined within a single lexical scope which destroys separate compilation and prevents reusing external classes for bridge objects (*e.g.,* no common iterator for different list classes);

Inflexible The initial bridge object must always be the outermost enclosing class of the classes defining an aggregate, which is inflexible. Also, it seems unlikely that external objects can be attached as ombudsmen to an existing aggregate.

State confusion There is no support for distinguishing between private state of the initial bridge object and the aggregate's representation. However, subsequent bridge objects may have private state.

Ad Hoc encapsulation Boyapati's bridge objects can be exported arbitrarily high up in the nesting hierarchy, making it hard to reason about the origins of changes and completely destroying strong encapsulation.

The strength of Boyapati's proposal is the ability to allow bridge objects to escape arbitrarily outside their defining aggregate. The downside of this flexibility is lack of flexibility in all other domains and the unclear guarantees that this built-in back door gives the system.

Lu et al. [20] overcome some of the limitations of Boyapati's escaping inner classes by allowing dynamically exposing internal representation through a "downgrading" operation. This voids the need of a specific inner class for exposure, which allows separate compilation and reuse, but just like with inner classes, their downgrading operation destroys the strong notion of encapsulation resulting in unclear properties of the resulting system. Shallow ownership (*e.g.,* [2]) is reminiscent of downgrading in that an object internal to an aggregate can arbitrarily pass on permission to reference aggregate objects to an external object that it creates. Shallow ownership however has no strong (or clear) encapsulation guarantee.

Ownership Domains [1] allows a programmer to manually specify contexts and how objects in these contexts may refer to each other by linking them. For instance, a list class may define a public domain for its iterators, which is linked to both the domain containing the list links and the element domain. While this is straightforward and gives plenty of flexibility ownership is not guaranteed by the system but must be manually encoded by the programmer. Ownership domains suffer from problems similar to Boyapati's inner classes in that public domains are publicly accessible, and therefore an iterator may be arbitrarily exported.

CoBoxes [26] (and JCoBoxes [27]) are active-objects-like systems with asynchronous message sends and futures. CoBoxes are similar to our aggregates in that they are defined in terms of the objects they contain and may have multiple entry-points into an aggregate. However, neither CoBoxes nor JCoBoxes rely entirely on runtime checks to protect a box's innards, whereas our system can express and check fortified aggregates with multiple entry points at compile-time.

MOJO [6] and Mojojojo [19][1] support multiple ownership, which is more general than our proposal. They can express multiple entry-points into an aggregate, but the general nature of the system comes at a cost of high complexity. Furthermore, (even though the papers say very little about alias restrictions) multiple ownership in these systems is a property of the type of the aggregate and the construction of aggregates may be done from the outside. In our system the aggregate owner is only visible inside the ombudsmen. Thus, in our system, aggregate ownership, population and extension is handeled from the inside only, by the aggregate itself.

In the context of verification of object invariants, Barnett and Naumann [3] define a friendship protocol in which a granting class can give privileges to another friend class that allows invariants in the friend to depend on fields in the granting class. Objects are connected using an explicit attach construct, but there is no notion of collaboratively defined state, and once a value of a field in a granting class has been obtained by a friend, the value may be exported arbitrarily.

Last, Joe$_1$ by Clarke and Drossopoulou [8] allow final variables to be used as owners to externally name an object's representation. This relaxation is however only made for variables on the stack and therefore cannot be used to express multiple entry points to aggregates in a straightforward fashion.

5. Discussion

5.1 Owners as Ombudsmen

Owners-as-ombudsmen is a straightforward extension to owners-as-dominators: the owners-as-dominators property holds for all objects in the **rep** context; objects inside **aggregate** contexts are instead dominated by an unknown subset of objects of the directly enclosing context. Thus, objects inside an aggregate context enjoy a weaker encapsulation than objects that belong to a private representation which is precisely the intention of our proposal since many aggregates cannot be expressed in the hierarchical fashion that deep ownership types dictate.

5.2 Ombudsmen and Reuse

Internally, an object will not know whether it lives inside another object's private representation, or constructs an aggregate, which allows programmers to design objects without concern for how they will be used in future systems. Consequently, an object cannot know whether it is dominated by *a single* object or a *collection of several* objects (which would presumably violate abstraction), but we have yet to see programming patterns where this is an important factor.

A small drawback of our system is that ombudsmen must be explicitly programmed as such since they must make use of the **aggregate** context. Removing this restriction is simple, just allow bridge objects to be given permission to reference aggregate objects, and involves the addition of a single type rule:

$$\text{(P-OMBUDSMAN)}\quad \frac{\vdash E}{E \vdash \texttt{bridge} \rightarrow^{\mathsf{ok}} \texttt{aggregate}}$$

This causes a problem with presenting a type of an ombudsman external to the aggregate, since there is no external name for the aggregate context. This can be solved using "lost owners" á la Generic Universe Types [12] *i.e.,* $\texttt{C}\langle\texttt{bridge}, \texttt{aggregate}\rangle$ will externally be $\texttt{C}\langle\texttt{owner}, ?\rangle$ where ? is an owner that cannot be named in the current context.

5.3 Ombudsmen and Existing Ownership Systems

The idea of ombudsmen was conceived during our work on extending Joëlle [10], a language for safe, reliable and efficient parallel programming based on the active object pattern [15]. To achieve the necessary isolation for active objects, Joëlle relies on an "flat" ownership types system where ownership forms a forest and every active object is a root in a tree in the forest.

Our extension to Joëlle sports a type and effect system which is an amalgamation of Greenhouse and Boyland's OOFX [16] and Clarke and Drossopoulou's Joe$_1$ together with support for externally unique (from Wrigstad's Joline [9, 28]), immutable and safe [23] references.

The ombudsman concept is easily combined with the concepts of our our extension to Joëlle, and can make good use of them. In Section 2.4, we showed how external uniqueness can be used to allow the external creation of a bridge object for an aggregate. Owner-polymorphic methods such as found in Clarke's dissertation or Joline can be used to temporarily export objects inside an aggregate context past their ombudsmen, but only for the duration of a method call.

When combining ombudsmen with an ownership-based effect system, such as the one in Joe$_1$ or our extension to Joëlle, the obvious question arises, how to report an effect to an object in the shared aggregate? The answer to the question is to externally report effects under the shared aggregate as effects to **owner**. This is imprecise, as not all objects in the **owner** context may access the aggregate. Without additional machinery, like path dependent-types (see *e.g.,* [8]), regions (see *e.g.,* [16]) or data groups [18], distinguishing ombudsmen for different aggregates is in any case impossible, so the subsuming **aggregate** into **owner** for effects is required for soundness.

6. Concluding Remarks

We have presented an extension to Clarkean ownership types that slightly relaxes owners-as-dominators to enable multiple entry points into a single aggregate. Our extension seems to work well with existing deep ownership systems, and only requires two additional ownership contexts, **aggregate** and **bridge**, and minor extensions to existing type rules. In terms of increasing complexity for the programmer, we believe that multiple contexts of an object does

[1] Although MOJO and Mojojojo differ in expressiveness and technical details they are very similar in spirit.

not overly complicate programming, especially since the contexts are limited to two, and the notion of owner specialisation, `bridge` is a subset of `owner`, should be as straightforward as any simple notion of regions.

References

[1] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, volume 3086 of *LNCS*, pages 1–25. Springer, June 2004.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, Nov. 2002.

[3] M. Barnett and D. A. Naumann. Friends Need a Bit More: Maintaining Invariants Over Shared State. In *Mathematics of Program Construction*, 2004.

[4] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston. Object-oriented analysis and design with applications, third edition. *SIGSOFT Softw. Eng. Notes*, 33:11:29–11:29, August 2008.

[5] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.

[6] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[7] D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[8] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 292–310, New York, NY, USA, 2002. ACM.

[9] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743/2003 of *Lecture Notes in Computer Science*, pages 59–67. Springer Berlin / Heidelberg, 2003.

[10] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal Ownership for Active Objects. In *Programming Languages and Systems (Procedings of the 6th Asian Symposium on Programming Languages and Systems)*, volume 5356/2008 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin / Heidelberg, 2008.

[11] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[12] W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Ph.D., Department of Computer Science, ETH Zurich, Dec. 2009. Doctoral Thesis ETH No. 18522.

[13] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In *In ECOOP*, pages 28–53. Springer, 2007.

[14] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Proceedings of Principles of Programming Languages (POPL)*, Jan. 2006.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] A. Greenhouse and J. Boyland. An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 205–229, London, UK, 1999. Springer-Verlag.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[18] K. R. M. Leino. Data groups: specifying the modification of extended state. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 144–153, New York, NY, USA, 1998. ACM.

[19] P. Li, N. Cameron, and J. Noble. Mojojojo - more ownership for multiple owners. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*. 2010.

[20] Y. Lu, J. Potter, and J. Xue. Ownership downgrading for ownership types. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 144–160, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Technical Report 263, Fernuniversität Hagen, 1999.

[22] P. Müller and A. Rudich. Ownership transfer in universe types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 461–478, New York, NY, USA, 2007. ACM.

[23] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *ECOOP*, 1998.

[24] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, Uniqueness, and Immutability. In *Objects, Components, Models and Patterns (Proceedings of 46th International Conference on Objects, Models, Components, Patterns)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer Berlin Heidelberg, 2008.

[25] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 311–324, New York, NY, USA, 2006. ACM.

[26] J. Schäfer and A. Poetzsch-Heffter. CoBoxes: Unifying active objects and structured heaps. In G. Barthe and F. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 201–219. Springer Berlin / Heidelberg, 2008.

[27] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In T. D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer Berlin / Heidelberg, 2010.

[28] T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Kista, Sweden, May 2006.