Multiple Aggregate Entry Points for Ownership Types*

Johan Östlund and Tobias Wrigstad

Uppsala University

Abstract. 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 present an extension of deep ownership that relaxes the single bridge object constraint 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; ombudsmen-sharing is explicit and 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 by clearly separating aggregation from the stronger composition, and further allows expressing common programming patterns such as iterators without resorting to systems that give unclear or unprincipled guarantees, or require additional complex machinery such as read-only references.

1 Introduction

Ownership types allow programmers to express encapsulation properties of programs in a compile-time checkable way. Ownership-based encapsulation has been used in many areas, including verification [22,27,31], reasoning about computational effects [13,34,15], information flow [3], memory management [9], object upgrades [8] and concurrent programming [6,17,15,39].

In classical "Clarkean" ownership types [12] every object belongs to another object and may be the owner of other objects. An *owned* object is said to belong to the *representation* of its owning object and cannot escape outside its owner. Consequently, an external object can only interact with a representation object via the public interface of its owner, which allows maintaining invariants and facilitates automated and manual reasoning. If we think of a heap as a graph of objects, all paths from the root of the graph to an object will include its owner—this is the *owners-as-dominators* property first formulated by Clarke et al. [16].

By imposing a hierarchical structure on the heap, owners-as-dominators gives strong and useful encapsulation guarantees, but at the cost of excluding some

* Supported in part by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 156–180, 2012.

common programming patterns. A common example of such a pattern is the iterator pattern for linked lists, which is the canonical example when introducing ownership types. A linked list should encapsulate its links, but for an iterator object to be able to access the next element in $O(1)$, it requires a direct reference to the "current" link in the list, which is only allowed in ownership systems if the iterator is *internal to the list itself.*

An alternative implementation that does not break the strong encapsulation of owners-as-dominators is to *unencapsulate* the list's links so that they become external to the list and therefore can be referenced by the iterator. This of course destroys the encapsulation.

In a nutshell: either the iterator can only be used inside the list (which renders it useless) or the list's encapsulation must be broken.

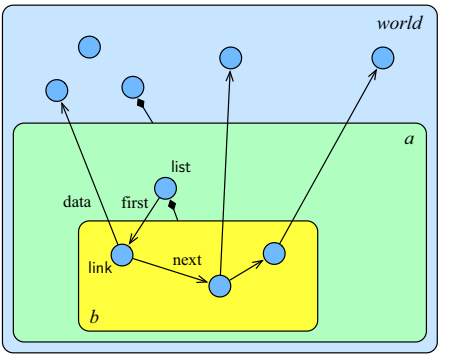
We can interpret a list as a software component with multiple service ports—one which implements the list interface and one which allows iterating over it. In this mindset, it makes sense to think of the links as encapsulated inside the (composite) component instead of inside some particular object that constructs it. However, implementing such a component in a Clarkean ownership system leads to exactly the above-mentioned problem of losing the encapsulation of the links: *there is no way to specify a set of objects shared between two or more objects that collaborate in defining a larger unit.* The problem is the unification of units of encapsulation and objects—the only way to introduce a unit K of encapsulation is by introducing a new object, which by virtue of owners-as-dominators blocks all direct access to the objects in K from external objects.

Contributions. This paper contributes to the field of Clarkean ownership systems by distinguishing between *composition* and *aggregation.* Just like traditional ownership systems, an object can be *composed* from representation objects which it dominates; additionally, an object may also *aggregate* other objects to which it may *share ownership* with other objects in a novel unit of encapsulation called an *aggregate.* Consequently, we can allow multiple entry points into an aggregate.

Our design allows programming idioms which rely on "principled sharing of representation" to be encoded in Clarkean ownership systems without compromising encapsulation by using aggregation instead of composition. The result encapsulation property is as easy to understand and almost as powerful as owners-as-dominators. Concretely, we:

- extend ownership types to support multiple entry points to a shared aggregate in a disciplined way with a strong encapsulation invariant called *ombudsmen-as-dominators*, which is clearly visible in the types;
- provide a simple and intuitive extension, adding only two new keywords;
- design the extension to be "pluggable"—it concerns only objects inside a shared aggregate, the semantics of old keywords is unmodified, and other objects enjoy strong encapsulation with owners-as-dominators;
- formalize the extension in a core language, and prove type soundness and our novel encapsulation invariant;
- have implemented our system on top of a Joline-like type checker; and
- provide an extensive coverage of related work.

```
class List<owner, data> {
  Link<this, data> first;
  void insert(Object<data> e) {
    Link<this, data> l =
                new Link<this, data>();
    l.data = e;
    l.next = first;
    first = l;
  }
}
class Link<owner, data> {
  Object<data> data;
  Link<owner, data> next;
}
```



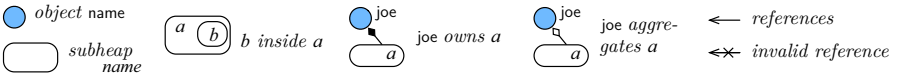


Fig. 1. *Left*: A list with ownership annotations. *Right*: Ownership structure of the linked list. The context a contains the linked list object which defines the context b for its representation objects (its links). Each link has a data field which points to its element objects. In this particular instance, the element objects reside in the outermost context `world`. The type of the list object is therefore `List<a,world>` binding the class' owner parameters `owner` and `data` to `a` and `world`, respectively.

2 Ombudsmen-as-Dominators

In this section we give an informal presentation of our system. First, however, we recap Clarkean ownership types as our encapsulation property *ombudsmen-as-dominators* builds directly on the classic notion of deep ownership types.

We use two main examples: iterators, with the dual purpose of introducing key concepts, and a shared bank account.

2.1 Clarkean Ownership Types

In Clarkean systems, the heap is hierarchically divided into a number of *contexts*, which can simply be thought of as sets of objects. Every object introduces a new context to hold its representation, nested inside the context where the object resides. In the source code, labels that denote run-time contexts are called *owners*. Owners are embedded in types to capture ownership and access permissions.

Linked Lists. Consider the linked list implementation shown in the left of Figure 1. Classes are parametrized over permissions to reference contexts; we call these *owner parameters*. The first owner parameter is always called `owner` and is special in that it also denotes the owner of the list instance, *i.e.,* the context where the list resides. The second parameter, called `data` in this example, is a necessary permission to reference the elements stored in the list.

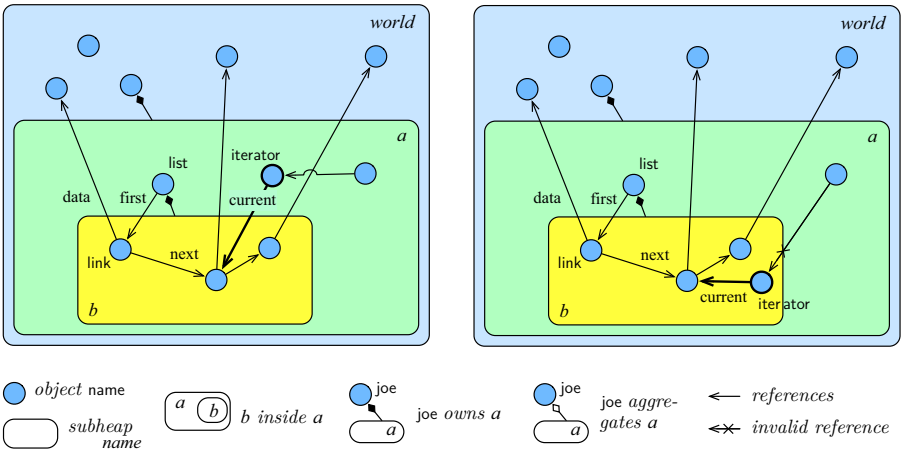


Fig. 2. Linked list with an iterator (showed with thicker lines). The iterator needs to reference the list's links, which breaks encapsulation. On the right, the iterator is moved inside the list shifting the problem to accessing the iterator.

The ubiquitous, implicitly declared owner **`this`** denotes the context introduced by the current object to hold its representation. In the list, all links are owned by **`this`** and are therefore encapsulated inside the list and cannot be exported outside. The `data` owner is forwarded to the links, as they too need permission to reference the element objects. In the `Link` class the `next` field has type `Link<`**`owner`**`,data>`, where **`owner`** is the list's representation.

The right hand side of Figure 1 depicts an instance of a heap with a list and a few contexts denoted by rounded boxes. The box b is the list's representation context, containing all its links, as they are owned by **`this`** in the `List` class. The list elements belong to the context **`world`**, which is the outermost context.

Owners-as-dominators allows references going outwards in the hierarchy, *e.g.,* the links may reference the elements, but not the converse. If an object x references an object y in a context k dominated by object z, then either $x = z$ or x must be inside k. In terms of Figure 1, if a reference crosses *into* a context, then the origin of the reference must be the object that owns the context.

The list implementation in Figure 1 makes sense from an object-oriented design point of view. The links are an implementation detail of the list and should not be observable from the outside. The problem with single entry point aggregates surfaces when we try to add an iterator to the `List` class, depicted in Figure 2 (left). Owners-as-dominators allows outward-going references, but the iterator needs to point inwards, into the list. The only way we can allow the iterator to reference the links is if we move the iterator into the list (context b), but then the iterator cannot be exported to a client of the list, Figure 2 (right).

The general problem is that ownership types cannot express two objects encapsulating a common context, for reasons made clear in the upcoming example.

```
class Person<owner, q> {
  Account<q> account; // private
  Person<owner, q> spouse;
  void share() {
    account = spouse.getAccount();
  }
  Account<q> getAccount() {
    return account;
  }
  ... // omitted
}
```

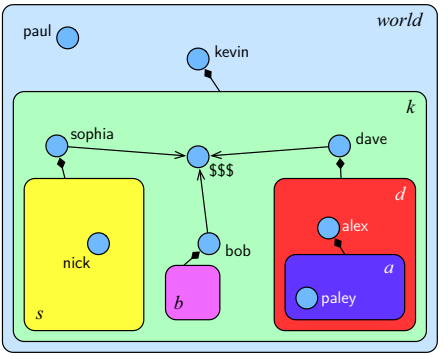


Fig. 3. Sophia and Dave sharing a bank account (**$$$**) in a system with owners-as-dominators. Bob also has access to the account. The legend can be found in Figure 2.

```
class Person<owner> {
  Account<aggregate> account;
  Person<bridge> spouse;
  void share() {
    account = spouse.getAccount();
  }
  Account<aggregate> getAccount() {
    return account;
  }
  ... // omitted
}
```

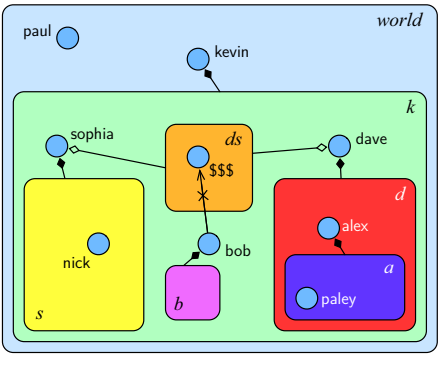


Fig. 4. Sophia and Dave sharing a *private* bank account in a system with owners-as-ombudsmen. The area *ds* denotes a shared context.

Shared Bank Account. Consider a bank account shared between two persons, sophia and dave. Let p be the owner of both sophia and dave, and q be the owner of the bank account. As should be clear by now, in Clarkean systems, the only way in which both sophia and dave can reference the account is if p is nested inside q, meaning that the account has *at most* the same level of encapsulation as sophia or dave. Figure 3 accomplishes this with the resulting ownership graph depicted to the right, which also includes a third person, bob.

For sophia and dave to be able to share an account, there must be a way for sophia to install (*e.g.,* a setter) a reference to the account in dave alternatively a way for dave to read the reference from sophia (*e.g.,* a getter). In either case, any object (*e.g.,* bob) with access to sophia or dave can access the shared account or install another one. Ombudsmen allow us to express the three objects as an aggregate where dave and sophia are in the aggregate's interface whereas the account is private. Figure 4 shows the code and resulting ownership graph.

Schäfer and Poetzsch-Heffter's work on CoBoxes [36,37] include more examples, *e.g.,* a DOM, file system, that need multiple aggregate entry points.

2.2 Ombudsmen

As Figure 4 hints, our proposal allows several objects—*ombudsmen*—to act as bridge objects, or entry points, between their common aggregate and the outside world. In terms of ownership types they share a common context. Objects in this context are "ombudsman-dominated"[1], 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.*

As the rightmost picture of Figure 3 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 *ombudsmen*-as-dominators, objects *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.

In the example in Figure 4 (right), sophia and dave are ombudsmen for a collaboratively defined aggregate containing a shared bank account. Their common aggregate context is depicted as the area *ds*. In addition, sophia and dave each have a representation context (*s* and *d*, respectively). The objects sophia and dave, as well as objects in their representation contexts, can reference objects in the aggregate context (*e.g.,* nick may reference $$$). Objects outside sophia or dave may not refer to objects in their aggregate context (*e.g.,* bob may not reference $$$). Objects in the aggregate context cannot reference the representation contexts of their ombudsmen (*e.g.,* $$$ may not refer to alex).

2.3 Typing Ombudsmen

We design the type system that lets us express aggregate encapsulation with multiple entry points as a relatively straightforward extension of Clarkean ownership types systems as found in *e.g.,* Joe$_1$ [13], Joline [14,38] or OGJ [35]. Classes are parametrized 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 represents 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 **`world`**, **`rep`**, **`aggregate`**, and **`bridge`**. The **`world`** keyword denotes the global outermost context. The **`rep`** keyword denotes the representation of the object and is equivalent to **`this`** in traditional ownership systems; **`aggregate`** denotes the aggregate context, which may be shared with other objects; 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 4, a field in sophia referencing dave (or the inverse) may have the owner **`owner`** or **`bridge`**. A reference from nick to

[1] We slightly abuse the term dominator to stay close to owners-as-dominators.

$$$ must have the owner **aggregate** (from the view of sophia, inside nick it is some other owner parameter which will be bound to sophia's **aggregate**); and sophia's reference to nick must have the owner **rep**.

Whether a field has owner **bridge** or **owner** makes an important difference. In the first case, we know that the field contains a reference to an object sharing the same aggregate. In the second case, we don't know if the reference points to an ombudsman for the same aggregate, or some other aggregate. In terms of our example, sophia could only ask for dave's reference to $$$ if sophia knows dave is a bridge object for *the same* aggregate. Otherwise, the reference might point to the representation of a different aggregate, which would break encapsulation.

Same Object, Different Types. Figure 5 shows an example of a Library component with two provided services with different privileges to access documents, `normalDocAccess` and `privilegedDocAccess`, and a required service `remoteLibrary`. To a client, the objects referred to by `normalDocAccess` and `privilegedDocAccess` are siblings to the component—they have the same owner. From the view of the client, the field `normalDocAccess` has type `AccessService<rep>` rather than `AccessService<bridge>` which would denote a bridge object of the aggregate in the client and not the component. For similar reasons, 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. The **bridge** owner identifies other ombudsmen of the *same* aggregate as the aggregate of the current object, which is therefore also an ombudsman.

Well-formed construction of aggregate objects is one of the key considerations of our system design. Any ombudsman has the capability to construct 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 of the shared aggregate are *siblings*. Coalescing an existing ombudsman object created externally with an aggregate is possible using ownership transfer [14,38,32]. In this case, one object must act as the "initial object" and move the unique objects into **bridge** (cf. Section 2.6).

Although we have defined ombudsmen for the Joe/Joline family of ownership systems [13,14,34,15], we believe they could easily be added to universe types [30,19,32] as well as OGJ [35], and similar.

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

2.4 Components

Standard UML components are implemented as aggregates of collaborating objects [5]. A component may provide several different interfaces (aka required and provided services); different applications may use different interfaces or a combination of different interfaces.

With ownership types, a component that wishes to export several different interfaces must do so through a single object if encapsulation is to be retained.

```
class Library<owner> {
  DocumentDB<aggregate> db = new DocumentDB<aggregate>;
  AccessService<bridge> normalDocAccess = new RestrictedPolicy<bridge>(db);
  AccessService<bridge> privilegedDocAccess =
                                    new UnrestrictedPolicy<bridge>(db);
  AccessService<owner>  remoteLibrary;
}

class Client<owner> {
  Library<rep> lib, otherLib;
  ...
  AccessService<rep> s1 = lib.normalDocAccess;
  c.remoteLibrary = s1; // = otherLib.normalDocAccess is also type sound
  AccessService<bridge> s2 = lib.normalDocAccess; // Fails!!
  lib.privilegedDocAccess = lib.privilegedDocAccess; // Fails!!
}
```

Fig. 5. Defining a component with two provided services and one required service

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, with the problems detailed on Page 160. To implement components with proper encapsulation, several objects must be able to share a common representation which cannot leak, as we did in Figure 4 and Figure 5.

2.5 Iterators with Ombudsmen

Figure 6 shows the ownership diagram for a linked list aggregate and Figure 7 the source code. Modulo the use of the novel **bridge** and **aggregate** owners, the code should be straightforward.

In the example, the list's links are part of the aggregate, and the list has no representation 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.

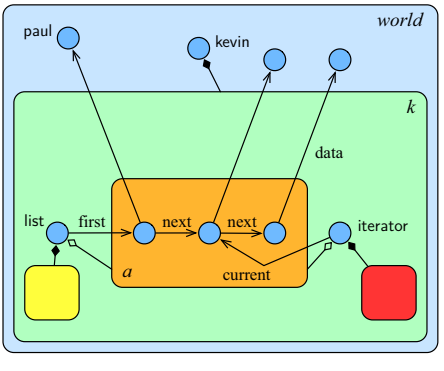


Fig. 6. Iterators with ombudsmen

Notably, in this solution, links are not considered part of the list object's representation, but part of the "list component's" aggregate context, which clearly reflects its degree of encapsulation.

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

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

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

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

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

Fig. 7. A list component with a (standard) list service and an iterator service

2.6 Staged Aggregate Construction

Allowing staged construction of aggregates is desirable as it allows *e.g.,* adding user-defined bridges to library aggregates. Attaching an additional bridge object O to an existing aggregate A will merge O's aggregate context, B say, with A.[2] This has consequences for all other bridges to B and unless we provide additional mechanisms for restricting merging of aggregate contexts, we have introduced a back door in the system. Going back to our bank accounts example, if bob can attach himself to dave and sophia's aggregate, then bob can suddenly call the getters and setters for the shared account, thereby gaining access to it.

Allowing the assignment from unique types to bridge types elegantly allows staged construction of an aggregate. Regular non-unique bridge objects cannot be attached to some other aggregate to gain access to its innards. Since an aggregate's **bridge** owner cannot be named externally, the aggregate implementer must explicitly provide a method to perform the attachment. Unless such a method is specified in any of the aggregate's bridges, staged construction is not possible. (This allows preventing bob from using the trick above.)

Aggregates can be constructed incrementally or in stages by attaching ombudsmen to each other, thereby merging their aggregate contexts. This practice 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.

Figure 8 (on Page 165) shows an excerpt of a `List` class that allows an object *external to the aggregate* to be made part of an existing aggregate. In the example, a unique iterator object is passed to the list's iterator method, which subsequently attaches it to its current aggregate by storing it in variable owned by **bridge**.

[2] And unless O is a sibling of A's bridge objects, the attachment is unsound since it would merge differently dominated aggregates.

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

Fig. 8. Attaching an external iterator to a list aggregate; -- is a destructive read

3 Formalizing Ombudsmen

In this section we formalize ombudsmen as an extension to deep ownership. The most relevant changes are in the type rules (EXPR-SELECT), (EXPR-UPDATE) and (EXPR-METHOD-CALL). Our formalism is inspired by [13,38].

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 $\mathtt{rep} \in \mathsf{Owners}(\tau) \Rightarrow e = \mathtt{this}$, 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. These can be seen in (EXPR-SELECT) and to some extent in (EXPR-UPDATE).

We consider only unary methods for simplicity and without loss of generality. Ombudsman adaptation is employed to translate internal types to external types and there is an additional visibility constraint that prevents calling methods which 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 is accomplished by adding an additional subtyping rule, see (BRIDGE-OWNER-SUBSUMPTION).

Conventions and Conveniences. We follow the practice of FJ [25] 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 $\overline{f} : \overline{\tau}$ 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 [16,12,14,38,34], we sometimes write $\mathtt{C}\langle\sigma\rangle$ for a type $\mathtt{C}\langle\overline{p}\rangle$ where σ is a map from the names of the formal parameters of C to the actual owner arguments $\overline{p}$. For example, if C is declared **class** $\mathtt{C}\langle\mathtt{owner}, \mathtt{a}, \mathtt{b}\rangle \cdots$ in the program, then if $\mathtt{C}\langle p_1, p_2, p_3\rangle$ is a well-formed type, we sometimes write $\mathtt{C}\langle\sigma\rangle$ for the implicitly defined $\sigma = \{\mathtt{owner} \mapsto p_1, \mathtt{a} \mapsto p_2, \mathtt{b} \mapsto p_3\}$. As a further convenience—following previous work—we sometimes write σ^p to mean $\sigma \cup \{\mathtt{owner} \mapsto p\}$ and σ_p to mean $\sigma \cup \{\mathtt{aggregate} \mapsto p\}$ (used in the dynamic semantics, possibly combined with σ^p).

3.1 Static Semantics

Syntax. The syntax of our system is defined in Figure 9. The meta variables x and y are used for names of variables (including **this**) and p and q for names

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

Fig. 9. Syntax

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 standard environment E, 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 judgments used in our static system.

Helper Predicates. A key helper predicate is OmbudsmanAdaptation, defined thus:

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

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

For every class C, we can derive a "field table" $\mathcal{FT}(\texttt{C})$ and a "method table" $\mathcal{MT}(\texttt{C})$. We define $\mathcal{FT}(\texttt{C})$ for $\texttt{class C}\langle\overline{p}\rangle\ \texttt{extends D}\langle\sigma\rangle\ \{\ F\ M\ \}$ as $F \bullet \sigma(\mathcal{FT}(\texttt{D}))$ and similar for $\mathcal{MT}(\texttt{C})$. $F(f) = \tau$ 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 $\bot$. Lookup in these tables is performed left–right, so $\mathcal{FT}(\texttt{C})(f)$ when $\mathcal{FT}(\texttt{C}) = F \bullet \sigma(\mathcal{FT}(\texttt{D}))$ is defined as $F(f)$ when $f \in \mathrm{dom}(F)$, else $\sigma(\mathcal{FT}(\texttt{D})(f))$. The root class has empty field and method tables; $\mathcal{FT}(\texttt{Object}) = \mathcal{MT}(\texttt{Object}) = \emptyset$ and $\emptyset(f) = \emptyset(m) = \bot$.

Using the tables, we define lookup helper predicates in a straightforward fashion where *first*, *second*, etc. extract the 1st, 2nd, etc. tuple compartments.

$$
\begin{array}{rcl}
\mathsf{FieldType}(\texttt{C}, f) & = & \mathcal{FT}(\texttt{C})(f) \\
\mathsf{Signature}(\texttt{C}, m) & = & \mathcal{MT}(\texttt{C})(m) \\
\mathsf{Param}(\texttt{C}, m) & = & \mathit{second}(\mathcal{MT}(\texttt{C})(m)) \\
\mathsf{Body}(\texttt{C}, m) & = & \mathit{third}(\mathcal{MT}(\texttt{C})(m)) \\
\mathsf{Fields}(\texttt{C}) & = & \{f \mid \mathcal{FT}(\texttt{C})(f) \neq \bot\}
\end{array}
$$

We also define functions for looking up owners from types: $\mathsf{Owners}(\texttt{C}\langle\overline{p}\rangle) = \{\overline{p}\}$ and $\sigma(\texttt{C}\langle\overline{p}\rangle) = \texttt{C}\langle\sigma(\overline{p})\rangle$.

Table 1. Judgments in the static system

$\vdash P : \tau$	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

Last, we assume the existence of a predicate $\mathsf{Arity}(\mathtt{C})$ that returns the number of owner parameters, including **owner**, declared for the class C, *e.g.,* $\mathsf{Arity}(\mathtt{List}) = 2$ from the example in Figure 7.

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.

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

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.

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

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

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

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

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

Expressions. Expressions are typed given the 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.

$$\frac{E \vdash e' : \tau' \quad E, x : \tau' \vdash e : \tau}{E \vdash \mathtt{let}\ x = e'\ \mathtt{in}\ e : \tau}\ (\textsc{expr-let}) \qquad \frac{\vdash E \qquad E(x) = \tau}{E \vdash x : \tau}\ (\textsc{expr-var})$$

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. [16], and that aggregate objects may only be accessed through ombudsmen. Last, we apply the OmbudsmanAdaptation helper function that make ombudsmen appear as regular objects when viewed externally.

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

(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 wild-cards in Java generics) and adds an additional restriction: a field holding an ombudsman can only be accessed through another ombudsman.

$$\frac{\begin{array}{c}E \vdash x : \mathtt{C}\langle\sigma^p\rangle \qquad \mathsf{FieldType}(\mathtt{C}, f) = \tau \qquad E \vdash y : \sigma^p(\tau) \\ \mathtt{rep} \in \mathsf{Owners}(\tau) \Rightarrow x = \mathtt{this} \\ \mathtt{bridge}, \mathtt{aggregate} \in \mathsf{Owners}(\tau) \Rightarrow p = \mathtt{bridge}\end{array}}{E \vdash x.f = y : \sigma^p(\tau)}\ (\textsc{expr-update})$$

The 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" (in the type system's view).

$$\frac{\begin{array}{c}(\textsc{expr-method-call})\\ E \vdash x : \mathtt{C}\langle\sigma^p\rangle \qquad \mathsf{Signature}(\mathtt{C}, m) = \tau_1 \to \tau_2 \qquad E \vdash y : \sigma^p(\tau_1)\\ \mathtt{rep} \in \mathsf{Owners}(\tau_1) \cup \mathsf{Owners}(\tau_2) \Rightarrow x = \mathtt{this}\\ \mathtt{bridge}, \mathtt{aggregate} \in \mathsf{Owners}(\tau_1) \Rightarrow p = \mathtt{bridge}\\ \mathtt{aggregate} \in \mathsf{Owners}(\tau_2) \Rightarrow p = \mathtt{bridge}\\ \mathsf{OmbudsmanAdaptation}(p, \tau_2) = \tau\end{array}}{E \vdash x.m(y) : \sigma^p(\tau)}$$

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

$$\frac{E \vdash \tau}{E \vdash \mathtt{null} : \tau}\ (\textsc{expr-null}) \qquad \frac{E \vdash \tau}{E \vdash \mathtt{new}\ \tau : \tau}\ (\textsc{expr-new}) \qquad \frac{E \vdash e : \tau' \qquad E \vdash \tau' <: \tau}{E \vdash e : \tau}\ (\textsc{expr-subsumption})$$

Type Environment Construction. We use a standard static type environment E.

$$\frac{}{\vdash \epsilon}\ (\textsc{e-}\epsilon) \qquad \frac{E \vdash \tau \qquad x \notin dom(E)}{\vdash E, x : \tau}\ (\textsc{e-var}) \qquad \frac{E \vdash q \qquad p \notin dom(E) \qquad \mathcal{R} \in \{\prec^*, \succ^*\}}{\vdash E, p\ \mathcal{R}\ q}\ (\textsc{e-context})$$

Contexts. Statically, contexts are added to the environment in (wf-class). The only manifest owner is `world`.

$$\frac{\vdash E \qquad p \in dom(E)}{E \vdash p}\ (\textsc{good-context}) \qquad \frac{\vdash E}{E \vdash \mathtt{world}}\ (\textsc{good-world})$$

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

$$\frac{\vdash E \qquad p \prec^* q \in E}{E \vdash p \prec^* q}\ (\textsc{inside}) \qquad \frac{\vdash E \qquad q \succ^* p \in E}{E \vdash p \prec^* q}\ (\textsc{outside}) \qquad \frac{E \vdash p}{E \vdash p \prec^* p}\ (\textsc{inside-reflexive}) \qquad \frac{E \vdash p \prec^* p' \qquad E \vdash p' \prec^* q}{E \vdash p \prec^* q}\ (\textsc{inside-transitive})$$

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).

$$\frac{E \vdash p \prec^* q}{E \vdash p \to^{\mathsf{ok}} q}\ (\textsc{p-inside}) \qquad \frac{\vdash E \qquad p \in \{\mathtt{bridge}, \mathtt{aggregate}\}}{E \vdash \mathtt{rep} \to^{\mathsf{ok}} p}\ (\textsc{p-rep})$$

An ombudsman's 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.

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

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.

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

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

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.

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

As an example of the use of this practice, see the list iterator example. Internally, the list's view of its iterator is $\mathtt{Iterator}\langle\mathtt{bridge}, \mathtt{data}\rangle$, but when obtained from some external object, the iterator's type is $\mathtt{Iterator}\langle\mathtt{owner}, \mathtt{data}\rangle$. This is sound since **bridge** always denotes a subset of **owner**.

3.2 Dynamic Semantics

The dynamic semantics is a big-step operational semantics. To distinguish diverging computation from stuck states, we use a standard trick to limit stack space [21,38]. Each reduction carries the remaining stack space and each method call reduces this number. A method call when there is no remaining stack space triggers an error.

Objects are represented by triples of type, aggregate context id α, and field mappings. Run-time types are the same as static types, but static owner names are substituted for run-time contexts. Run-time contexts are κ (an object id ι, aggregate context id α, or the special context **world**). Values are ι or ϵ (null).

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

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 and bindings, plus the initial 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.

$$\frac{\langle H; B; e\rangle \ \rightarrow_n \ \langle H'; v'\rangle \quad \langle H'; B, x \mapsto v'; e'\rangle \ \rightarrow_n \ \langle H''; v''\rangle}{\langle H; B; \texttt{let}\ x = e\ \texttt{in}\ e'\rangle \ \rightarrow_n \ \langle H''; v''\rangle}\ (\text{D-LET}) \qquad \frac{B(x) = v}{\langle H; B; x\rangle \ \rightarrow_n \ \langle H; v\rangle}\ (\text{D-VAR})$$

Looking up a field on an object receiver is straightforward. We write $H(\iota.f)$ as a shorthand for $\mathcal{F}(f)$ when $H(\iota) = (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F})$. Field updates are similar, and we write $H(\iota.f) := v$ for $H[\iota \mapsto (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F}[f \mapsto v])]$ when $H(\iota) = (\mathtt{C}\langle\overline{\kappa}\rangle, \alpha, \mathcal{F})$.

$$\frac{B(x) = \iota \qquad H(\iota.f) = v}{\langle H; B; x.f\rangle \ \rightarrow_n \ \langle H; v\rangle}\ (\text{D-SELECT}) \qquad \frac{B(x) = \iota \qquad B(y) = v}{\langle H; B; x.f = y\rangle \ \rightarrow_n \ \langle H(\iota.f) := v; \epsilon\rangle}\ (\text{D-UPDATE})$$

In our simple semantics, method calls cause the evaluation of a method body under a new B with all owner names substituted for their run-time equivalents, derived from the current `this`. Furthermore, `this` is substituted for the current object, and the parameter is substituted for the actual argument value.

$$\frac{\begin{array}{c} B(x) = \iota \qquad B(y) = v \qquad H(\iota) = (\mathtt{C}\langle\sigma^{\kappa}\rangle, \alpha, \_) \\ \mathsf{Body}(\mathtt{C}, m) = e \qquad \mathsf{Param}(\mathtt{C}, m) = x \\ B' = \texttt{rep} \mapsto \iota, \texttt{bridge} \mapsto \kappa, \texttt{this} \mapsto \iota, x \mapsto v, \texttt{aggregate} \mapsto \alpha \\ n > 0 \qquad \langle H; B', \sigma^{\kappa}; e\rangle \ \rightarrow_{(n-1)} \ \langle H'; v'\rangle \end{array}}{\langle H; B; x.m(y)\rangle \ \rightarrow_n \ \langle H'; v'\rangle}\ (\text{D-METHOD-CALL})$$

The run-time representation of `null` is denoted by ϵ. Object creation is simple due to the absence of constructors and custom field initialization. A fresh object has all its fields initialized 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.

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

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

3.3 Meta Theory

In our reasoning about well-formedness, we rely on a combined type environment and store type $\Gamma ::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, \iota : \tau \mid \Gamma, \alpha : \kappa \mid \Gamma, \mathtt{o}\,\iota : \alpha$. The entry $\alpha : \kappa$

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

$\vdash \Gamma$	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
$\Gamma \vdash \mathtt{C}\langle\overline{\kappa}\rangle$	$\mathtt{C}\langle\overline{\kappa}\rangle$ is a well-formed type under Γ
$\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \kappa'$	Objects in context κ have 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

maps an aggregate context α to the owner κ of all its ombudsmen. In a similar fashion, the entry $\mathtt{o}\,\iota : \alpha$ maps an object ι into an aggregate context α for which it acts as an ombudsman. Table 2 overviews the judgments in the meta theory.

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

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

$$\frac{\Gamma \vdash \kappa \qquad \alpha \notin dom(\Gamma)}{\vdash \Gamma, \alpha : \kappa}\ (\Gamma\text{-BRIDGE}) \qquad \frac{\vdash \Gamma \qquad \mathtt{o}\,\iota \notin dom(\Gamma) \qquad \alpha : \kappa \in \Gamma \qquad \Gamma(\iota) = \mathtt{C}\langle\sigma^{\kappa}\rangle}{\vdash \Gamma, \mathtt{o}\,\iota : \alpha}\ (\Gamma\text{-AGGREGATE})$$

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.

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

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

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

We assume a function $e/\tau\,\{B\}$ that replaces static names of owners in the domain of B with their dynamic counterparts, *e.g.,* $\mathtt{C}\langle p\rangle\,\{B\} = \mathtt{C}\langle B(p)\rangle$. The judgments

$\Gamma \vdash e : \tau$ are "copy-and-patch" from the corresponding type rules $E \vdash e : \tau$ and therefore omitted.

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \epsilon : \tau}\ (\textsc{null-type}) \qquad \frac{\Gamma(\iota) = \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \iota : \tau}\ (\textsc{object-type})$$

Rules for good dynamic contexts are similar to their static counterparts. A type is well-formed if its owner has the right to reference all other owner parameters.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{world}}\ (\textsc{context-world}) \qquad \frac{\vdash \Gamma \qquad \kappa \in dom(\Gamma)}{\Gamma \vdash \kappa}\ (\textsc{context-object/aggregate}) \qquad \frac{\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \overline{\kappa} \qquad \mathsf{Arity}(\mathtt{C}) = |\kappa, \overline{\kappa}|}{\Gamma \vdash \mathtt{C}\langle \kappa, \overline{\kappa} \rangle}\ (\textsc{d-type})$$

Except 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.

$$\frac{\vdash \Gamma \qquad \Gamma(\kappa) = \mathtt{C}\langle \overline{\kappa} \rangle \qquad \kappa' \in \overline{\kappa}}{\Gamma \vdash \kappa \prec^* \kappa'}\ (\textsc{d-inside}) \qquad \frac{\vdash \Gamma \qquad \Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \prec^* \kappa}\ (\textsc{d-ombudsman})$$

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.

$$\frac{\Gamma \vdash \kappa \prec^* \kappa' \quad \vee \quad \Gamma \vdash \kappa \prec^* \iota \wedge \Gamma(\mathtt{o}\,\iota) = \kappa'}{\Gamma \vdash \kappa \rightarrow^{\mathsf{ok}} \kappa'}\ (\textsc{d-mayref})$$

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 $\iota \sqsupseteq \iota' \iff H(\iota) = (\mathtt{C}\langle \overline{\kappa} \rangle, \alpha, \_) \wedge H(\iota') = (\mathtt{C}\langle \alpha, \_ \rangle, \_, \_)$. We can now define "ombudsmen-as-dominators" as a straightforward extension to owners-as-dominators.

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

> In plain English this states that if an object ι_1 references another object ι_2, then either the owner of ι_2 is a dominator of ι_1 (this is the standard owners-as-dominators property, the non-savvy ownership reader can consult *e.g.,* Clarke's dissertation [12] for additional details), or ι_2 is part of some aggregate and ι_1 is inside the representation of this aggregate.

Proof. Assume $\Gamma \vdash H$ in which $\iota_1 \rightarrow \iota_2$. Let the run-time type of ι_1 be some type $\mathtt{C}\langle \overline{k} \rangle$. By (HEAP-OBJECT), ι_2 is owned by some owner, k say, in $\overline{k}$, or the run-time values for **world**, **rep** (ι_1), **bridge** (ι_1's owner) and **aggregate** (α, say).

First, let k' be the owner of ι_1. By (D-INSIDE) follows $\iota_1 \prec^* k'$ and by (D-TYPE) follows $k' \rightarrow^{\mathsf{ok}} k$ for any k in $\overline{k}$. Now, by (D-MAYREF), either $k' \prec^* k$ (implying $\iota_1 \prec^* k$ by transitivity), or exists ι s.t. $\iota_1 \prec^* \iota$ (by transitivity) and $\Gamma(\mathtt{o}\,\iota) = k'$, which implies $\iota \sqsupseteq \iota_2$. (∗) If $k = \mathtt{world}$ then $\iota_1 \prec^* k$ by definition. If $k = \iota_1$, then $\iota_1 \prec^* k$ by reflexivity of $\prec^*$. The case when $k = i_1$'s owner is subsumed by the above since ι_1's owner is in $\overline{k}$. If $k = \alpha$, then second part of the theorem's implication holds for $\iota = \iota_1$ (∗∗)

For clarity, in (∗), ι_1 belongs to the representation of an object that aggregates ι_2. In (∗∗), ι_1 aggregates ι_2.

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 $\langle H; B; e\rangle \rightarrow_n \langle H'; v\rangle$ or $\langle H; B; e\rangle \rightarrow_n \mathsf{ERR}$ for some finite value of n.

Proof. The proof is straightforward by induction on the big-step rules where most cases are immediate. The slightly more intricate cases, (D-SELECT), (D-UPDATE) and (D-METHOD-CALL) are all guarded by versions of the same rule (elided in this presentation) that capture null-dereferencing or stack overflow. 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. □

PRESERVATION: If $\Gamma \vdash \langle H; B; e\rangle : \tau$ and $\langle H; B; e\rangle \rightarrow \langle H'; v\rangle$, then there exists $\Gamma' \supseteq \Gamma$ s.t. $\Gamma' \vdash \langle H'; B; v\rangle : \tau$ (omitting stack space for simplicity).

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

4 Related Work

Several researchers have proposed relaxations of Clarkean ownership types that can be used to overcome the single bridge object-problem. The main difference with these systems is that ombudsmen explicate the notions of aggregates and bridges in the types, *e.g.,* allowing an object to clearly identify other bridges to its aggregate, and work within the confines of owners-as-dominators, rather than providing a "back door" which allow external access to an object's internals.

Boyapati et al. [7] 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 back door to an object's representation exists 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. This destroys separate compilation, and also prevents reusing external classes for bridge objects (*e.g.,* it is not possible to have a common iterator class for different list classes).

Inflexible. The initial bridge object must always be the outermost enclosing class of the classes defining an aggregate. This is inflexible as it does not allow defining an aggregate which can be created in multiple ways, using different classes (*e.g.,* a component with different ports for different configurations). Also, staged construction of aggregates is not possible.

State confusion. There is no support for distinguishing between the representation of the initial bridge object (private implementation details) and the aggregate's representation (which might be exported to another bridge). Consequently, an iterator can leak details about the list which were not intended to be exported. Strangely enough, the iterator can have representation which is private from the list.

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.

OIGJ [40] take Boyapati's approach and suffers from all but the last problem above. In OIGJ, inner classes have access to the representation of the enclosing object. The difference to Boyapati's system is that in OIGJ the inner class must have the same owner as the outer object (*e.g.,* iterator has same owner as its list), and thus cannot be arbitrarily exported, similar to our system. There is however no way to distinguish bridges of a common aggregate from other peers. As in Boyapati's system the outer object grants subsequent bridge objects access to all its state, there is no distinguishing of aggregate and bridge object representation.

The encapsulation property of Universe Types [30], 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 to the list elements via an iterator (modulo expensive and unsafe downcasts). Generic Universe Types [19] 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. In a concurrent setting this may not be desirable as read-only references do not preclude the existence of mutable aliases, making

them subject to possible data races. Clarke et al. [15] similarly relax owner-as-dominators but for *safe* references (that may only be used to read immutable parts of objects), and for references to immutable data, guaranteeing that no mutation may occur concurrently.

Lu et al. [29] 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 together. 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 flexible, it is difficult to identify the encapsulation invariants of a system: it is necessary to look at large parts of a system and how its components introduce new links between contexts that would invalidate assumptions about encapsulation drawn locally by just studying the list class. Ownership domains further 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 [36] (and JCoBoxes [37]) 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, both CoBoxes and JCoBoxes rely entirely on run-time checks to protect a box's innards, whereas our system can express and check fortified aggregates with multiple entry points at compile-time.

MOJO [11] and Mojojojo [28][3] support multiple ownership, which is more general than our proposal, but this flexibility comes with very high complexity. The descriptive nature of MOJO and Mojojojo allows a programmer to express an aggregate in the types, but encapsulation is not enforced. Further, because the aggregate is visible (in the types) from the outside it can be constructed, populated and extended from the outside. In our system such operations are under complete control of the aggregate itself.

In the context of verification of object invariants, Barnett and Naumann [4] 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

[3] Although MOJO and Mojojojo differ in expressiveness and technical details they are very similar in spirit, so we treat them as one here.

in a granting class has been obtained by a friend, the value may be exported arbitrarily.

Boyland et al. [10] use effects and a novel "from" annotation to allow a data structure to temporarily yield its state to an external object (*e.g.,* a list to an iterator). Such access permissions are treated linearly, and therefore cannot be used to express multiple (read/write) entry points to an aggregate, including fail-fast iterators with mutating capabilities and our shared bank account.

Lastly, Joe$_1$ by Clarke and Drossopoulou [13] 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

Ombudsmen-as-dominators 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 representation objects which is precisely the intention of our proposal since many aggregates cannot be expressed in the hierarchical fashion that deep ownership types dictate. This has consequences for reuse and computational effects, which is discussed below.

5.1 Ombudsmen and Reuse

Internally, an object will not know whether it lives inside another object's 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 not yet seen a programming pattern where this is an important factor.

A drawback of our system is that a class cannot be retrofitted to be an ombudsman unless it makes use of the **aggregate** context. Removing this restriction is simple, just give bridge objects implicit permission to reference aggregate objects, and involves the addition of a single type rule:

$$\frac{\vdash E}{E \vdash \texttt{bridge} \rightarrow^{\mathsf{ok}} \texttt{aggregate}}\ (\textsc{p-ombudsman})$$

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" [18]. The type C⟨**bridge**, **aggregate**⟩ will externally be C⟨**owner**, ?⟩ where ? is an owner that cannot be named in the current context.

5.2 Ombudsmen and Ownership-Based Computational Effects

The idea of ombudsmen was conceived during our work on extending Joëlle [15], a language for safe, reliable and efficient parallel programming based on the active object pattern [23]. 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 of 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 [24] and Clarke and Drossopoulou's Joe_1 together with support for externally unique (from Wrigstad's Joline [14,38]), immutable and safe [33] references.

In Section 2.6, we showed how external uniqueness can be used to allow the external creation of a bridge object for an aggregate without introducing back doors. Owner-polymorphic methods, such as found in Clarke's dissertation [12] or Joline [38], 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 of 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.,* [13]), regions (see *e.g.,* [24]), linearity (see *e.g.,* [10]) or data groups [26], 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 works 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 not overly complicate programming, especially since the contexts are limited to two, and the notion of owner specialization, `bridge` is a subset of `owner`, should be as straightforward as any simple notion of regions.

We have implemented the ombudsman system as part of our extended Joëlle compiler on top of JastAddJ [20]. It currently supports deep ownership types, external uniqueness, and a complete implementation of ombudsmen, including staged aggregate construction with ownership transfer.

Acknowledgments. We thank the anonymous reviewers at IWACO 2011 and the ECOOP 2012 reviewers for their valuable feedback which helped us improve our presentation, and correct mistakes.

References

1. Aldrich, J., Chambers, C.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
2. Aldrich, J., Kostadinov, V., Chambers, C.: Alias Annotations for Program Understanding. In: OOPSLA (November 2002)
3. Banerjee, A., Naumann, D.A.: Secure Information Flow and Pointer Confinement in a Java-like Language. In: Proceedings of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW), pp. 253–267. IEEE Computer Society Press (June 2002)
4. Barnett, M., Naumann, J.D.A.: Friends Need a Bit More: Maintaining Invariants Over Shared State. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
5. Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J., Houston, K.A.: Object-oriented analysis and design with applications, third edition. SIGSOFT Softw. Eng. Notes, 33, 11:29–11:29 (2008)
6. Boyapati, C., Lee, R., Rinard, M.: Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In: OOPSLA (2002)
7. Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. In: POPL (2003)
8. Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H., Richman, S.: Lazy Modular Upgrades in Persistent Object Stores. In: OOPSLA, pp. 403–417. ACM, New York (2003)
9. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In: PLDI (June 2003)
10. Boyland, J., Retert, W., Zhao, Y.: Iterators can be Independent from Their Collections. In: IWACO (2007)
11. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple Ownership. In: OOPSLA (2007)
12. Clarke, D.: Object Ownership and Containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia (2001)
13. Clarke, D., Drossopolou, S.: Ownership, Encapsulation and the Disjointness of Type and Effect. In: OOPSLA (2002)
14. Clarke, D., Wrigstad, T.: External Uniqueness is Unique Enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 176–201. Springer, Heidelberg (2003)
15. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal Ownership for Active Objects. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 139–154. Springer, Heidelberg (2008)
16. Clarke, D.G., Potter, J., Noble, J.: Ownership Types for Flexible Alias Protection. In: OOPSLA, pp. 48–64 (1998)
17. Cunningham, D., Drossopoulou, S., Eisenbach, S.: Universe Types for Race Safety. In: VAMP 2007, pp. 20–51 (September 2007)
18. Dietl, W.: Universe Types: Topology, Encapsulation, Genericity, and Tools. Ph.D., Department of Computer Science, ETH Zurich, Doctoral Thesis ETH No. 18522 (December 2009)
19. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)
20. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: OOPSLA, pp. 1–18. ACM, New York (2007)

21. Ernst, E., Ostermann, K., Cook, W.R.: A Virtual Class Calculus. In: Proceedings of Principles of Programming Languages (POPL) (January 2006)
22. Fahndrich, M., Xia, S.: Establishing Object Invariants with Delayed Types. SIGPLAN Not. 42(10), 337–350 (2007)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
24. Greenhouse, A., Boyland, J.: An Object-Oriented Effects System. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 205–229. Springer, Heidelberg (1999)
25. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (2001)
26. Leino, K.R.M.: Data groups: specifying the modification of extended state. In: OOPSLA, pp. 144–153. ACM, New York (1998)
27. Leino, K.R.M., Müller, P.: Object Invariants in Dynamic Contexts. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
28. Li, P., Cameron, N., Noble, J.: Mojojojo - more ownership for multiple owners. In: International Workshop on Foundations of Object-Oriented Languages, FOOL (2010)
29. Lu, Y., Potter, J., Xue, J.: Ownership Downgrading for Ownership Types. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 144–160. Springer, Heidelberg (2009)
30. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for controlling representation exposure. In: Poetzsch-Heffter, A., Meyer, J. (eds.) Programming Languages and Fundamentals of Programming, pp. 131–140. Technical Report 263, Fernuniversität Hagen (1999)
31. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular Specification of Frame Properties in JML. In: Concurrency and Computation Practice and Experience (2003)
32. Müller, P., Rudich, A.: Ownership Transfer in Universe Types. In: OOPSLA, pp. 461–478. ACM, New York (2007)
33. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
34. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, Uniqueness, and Immutability. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 178–197. Springer, Heidelberg (2008)
35. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic Ownership for Generic Java. In: OOPSLA, pp. 311–324. ACM, New York (2006)
36. Schäfer, J., Poetzsch-Heffter, A.: CoBoxes: Unifying Active Objects and Structured Heaps. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 201–219. Springer, Heidelberg (2008)
37. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
38. Wrigstad, T.: Ownership-Based Alias Management. PhD thesis, Royal Institute of Technology, Kista, Stockholm (May 2006)
39. Wrigstad, T., Pizlo, F., Meawad, F., Zhao, L., Vitek, J.: Loci: Simple Thread-Locality for Java. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 445–469. Springer, Heidelberg (2009)
40. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and Immutability in Generic Java. In: OOPSLA 2010, pp. 598–617. ACM, New York (2010)