Ownership Types: A Survey*

Dave Clarke[1], Johan Östlund[2], Ilya Sergey[1], and Tobias Wrigstad[2]

[1] iMinds-DistriNet, Department of Computer Science, KU Leuven, Belgium
[2] Department of Information Technology, Uppsala University, Sweden

Abstract. Ownership types were devised nearly 15 years ago to provide a stronger notion of protection to object-oriented programming languages. Rather than simply protecting the fields of an object from external access, ownership types protect also the objects stored in the fields, thereby enabling an object to claim (exclusive) ownership of and access to other objects. Furthermore, this notion is statically enforced by now-standard type-checking techniques.

Originating as the formalisation of the core of Flexible Alias Protection, ownership types have since been extended and adapted in many ways, and the notion of protection provided has been refined into topological and encapsulation dimensions. This article surveys the various flavours of ownership types that have been developed over the years, along with the many applications and other developments. The chapter concludes by suggesting some directions for future work.

Aliasing is endemic in object-oriented programming.

Noble, Vitek, Potter [112].

1 Introduction

Object aliasing is one of the key challenges that must be addressed when constructing large software systems using an object-oriented language. Bugs due to unintentional aliases are notoriously difficult to track down and can lead to unexpected side-effects, invalidated invariants, reasoning based on faulty assumptions, a wealth of security bugs, and more. On the other hand, shared mutable state and a stable notion of object identity are considered to be core ingredients of the object-oriented paradigm, and mutable object structures are frequently used to model real-world situations involving sharing. Dealing with aliasing, by either banning it, clearly advertising it or otherwise managing or controlling its effects, therefore has become a key research issue for object oriented-programming [77,78]. Mainstream object-oriented programming languages such as Java, C# and Scala provide no special means to simplifying working with

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu), KULeuven BOF Project STRT1/09/031: DesignerTypeLab and the Swedish Research Council within the UPMARC Linnaeus centre of Excellence and the project Structured Aliasing.

aliases beyond favouring references instead of pointers, and providing automatic garbage collection (although one Scala plugin does offer support for Uniqueness and Borrowing [71], see Section 6).

In 1998, Noble, Vitek and Potter [112] introduced Flexible Alias Protection as a conceptual model of inter-object relationships. Rather than banning aliasing altogether, the key idea was to limit the visibility of changes to objects via aliases. This was done either by limiting where an alias could propagate and by limiting the changes that could be observed though an alias. The ideas put forward in this work could be statically checked based on programmer-supplied type annotations called aliasing modes.

In order to better understand Flexible Alias Protection, Clarke, Potter and Noble [46] formalised its core mechanisms, resulting in Ownership Types. Although originating as the core of Flexible Alias Protection, the Ownership Types system made a few contributions beyond providing a clean formalisation. Firstly, Flexible Alias Protection did not offer enough machinery to provide a type for **`this`**; Ownership Types corrected this problem, thereby introducing the notion of owner—the owner of **`this`** (the current object) is **`owner`** (the owner of the current object). Secondly, Clarke, Potter and Noble provided a formalisation and proof of a topological property on object graphs enforced by the type system, namely the owners-as-dominators invariant. In their work, a program's heap is divided into hierarchically nested regions, originally called ownership contexts.[1] An ownership context is a set of objects. Every object belongs to a single context, and each object defines a context for its protected representation objects. An object A in the representation context defined by object B is said to be owned by B, and B is called the owner of A. Upon creation, each object was placed firmly in a single ownership context, its owning context, for its life-time. The object also receives a set of permissions to reference objects in other ownership contexts (a subset of those visible at the place of instantiation). Information about the owner of an object appears in its type—hence the name Ownership Types—and this is used to govern which parts of an object are accessible to other objects and when an object can be passed to other objects.

The original Ownership Types system was designed for a small language; for instance, it did not state how to deal with inheritance and subtyping, and it did not provide semantics for the remaining constructs of Flexible Alias Protection. Over the years these issues have been explored and different variations have been proposed. These variants impose different policies that are more flexible and less restrictive than owners-as-dominators; they introduce generalisations or other extensions and apply Ownership Types in different application scenarios. The goal of this article is to survey this, now vast, body of work.

Outline. The paper is organised as follows. Section 2 defines some of the basic concepts used in the Ownership Types literature. Section 3 presents a survey of the different kinds of topological restrictions and notions of encapsulation

[1] Originally the word context was used, but for uniformity of presentation, we will use ownership context or simply owner.

based on Ownership Types and related systems. Section 4 considers various extensions to the basic model, including generics, computational effects, dynamic ownership, and ownership transfer. Section 5 discusses the important issue of ownership inference, as ownership types typically impose a significant syntactic overhead. Section 6 gives an overview of applications of Ownership Types. Section 7 briefly discusses some of the foundational work done on Ownership Types and its variants. Section 8 explores some of the empirical studies done on ownership in larger code bases. Finally, Section 9 presents a discussion of future directions for research on Ownership Types and concludes the survey.

2 Groundwork

This section define some concepts required to understand Ownership Types and its variants. To be consistent within this survey, we have tried to present uniform terminology rather than reuse the terms given in the original research papers.

Ownership Types systems work in two ways to restrict object graphs underlying the run-time heap of object-oriented programs. The first is by providing *topological restrictions* on the reference structure of the object graph. The second approach is to enforce *encapsulation*, which occurs by limiting operations that can be performed via certain references in the object graph so that the places where mutation of objects can occur are restricted in scope.

The core concept of any Ownership Types system is *object ownership*, namely that objects are owned by other objects or perhaps other entities (global owners, stack-based owners, ...). An *Ownership Types system* is a type system where types are annotated or otherwise associated with information about object ownership.

An *ownership context* is a region of the heap or a collection of objects. More informally, it is a box into which objects are placed [61]. These boxes are generally organised hierarchically: all objects have boxes to store (and protect) the objects they own, and each object is considered to be *inside* the object whose box it is placed in. The hierarchy induces a *nesting* relationship between objects. If the ownership context corresponds to an object, typically this object is referred to as the *owner*. In the literature ownership contexts are also called ownership domains [4], contexts [46], boxes [33], and regions [25].

The objects residing in the ownership context of some object are called the *representation* of the object. From a semantic point of view, the representation of an object consists of the objects that contribute to the implementation of the abstraction that the object represents. Being able to directly access the representation of an object, and thereby violate the invariants of that object, is called *representation exposure*. Some Ownership Types systems prevent access to representation objects (a topological restriction). Other systems allow in addition objects to be logically contained in an object [4], but impose no topological restrictions. Yet other systems allow access to representation objects, but only via references with limited capabilities (thereby enforcing encapsulation).

Objects in the same ownership context are called *siblings* or *peers*.

In the type system, classes may have parameters which will be supplied with different owners when the class is instantiated, thereby allowing *owner polymorphism.* These parameters are called *owner parameters.* The syntactic mechanism underlying ownership parameterisation is analogous to type parameterisation. The owner of an instance is indicated using one such parameter; this parameter is often implicit in the class header and is generally referred to using keyword **`owner`** within the class body. Parameterisation can occur at the level of methods, resulting in owner polymorphic methods.

Ownership Types systems generally have an ownership context into which *shared objects* are placed. Generally, this ownership context is accessible to all other objects. In various systems this is known as **`shared`**, **`world`** and, originally, **`norep`**. *Manifest ownership* occurs when the owner of all instances of a class is defined to be a particular, typically globally known, owner. Manifest ownership occurs when a class cannot be parameterised.

One extreme way of controlling aliasing is to remove aliasing all together—this is a topological constraint.

A *unique reference* to an object is the only reference to the object in the system. Various weakenings of this notion exist. For instance, an *externally unique* reference is the only reference to an object that is not (transitively) inside the object's representation. Sometimes multiple references to an object exist, but only one of them can be used. At other times, there may be multiple references to an object, but only one reference can be used to mutate the object. When an object is created, it is typically considered to be *free* (sometimes called *virgin*), meaning that there are no references to it in fields. Similarly, when a field uniquely referring to an object releases that object (perhaps via a nullifying destructive read), then the object can again be considered free. A unique reference or a reference to some owned object may be temporarily passed to another object, generally for the duration of a method call, so that when the method is over all temporary references vanish or become unusable. The references are call *borrowed* or *lent* references and the process is known as *borrowing.*

An alternative way of controlling aliasing is to control the effects of aliasing—this imposes encapsulation.

Immutability means that the state cannot change. An *immutable object* or *value object* is one whose fields cannot change. Some objects have immutable slices, meaning that only some of the fields are immutable. Most work on immutability only considers an object to be immutable if all of its fields reference only immutable objects or primitive values, *i.e.,* immutability is transitive, though variants consider references to (external) mutable objects, with restrictions imposed on how the immutable objects can depend on the mutable objects. Another way of putting this is that immutability is only transitive for owned immutable objects. A *pure method* is a method that does not update any fields of any objects. More refined versions of this notion may be allowed to update object fields, so long as these updates are not observable outside of the method. An *impure method* is a method that is not pure. A *read-only reference* is a reference through which only pure methods can be called. One property of read-only references is that they can observe changes to the object state, resulting in what is called *observational exposure*. A stronger variant of read-only reference is possible, namely one through which only immutable state can be read.

3 Models Restricting Topology and Enforcing Encapsulation

This section discusses Ownership Types systems and the kinds of protection enforced by the systems in terms of topological restrictions and encapsulation discipline. Figure 1 demonstrates some of the core constraints imposed by various Ownership Types disciplines. Owners-as-dominators, which imposes a hierarchical structure on the heap, is the strictest discipline, allowing only a single entry point to the internal objects. This is historically the first topological invariant enforced by an Ownership Types system, and will be discussed first. This has been relaxed in various ways, such as by allowing proxy objects to access internal representation objects, thus weakening the topological restrictions, or by allowing references with less capabilities to access internal objects. After introducing owners-as-dominators, we move on to discussing relaxations of this strong property for stack- and heap-based aliases, and then to systems which take a fundamentally different approach to specifying topological/encapsulation policies, such as owners-as-modifiers, ownership domains and multiple ownership. Finally, we present an overview of the confined types approach, which is syntactically much simpler than other systems, at the cost of a less expressive notion of protection.

3.1 Owners-as-Dominators

Owners-as-dominators is the topological invariant enforced by the original Ownership Types system of Clarke et al. [46]. In this ownership system, an object could be given explicit permission to reference the direct representation of any

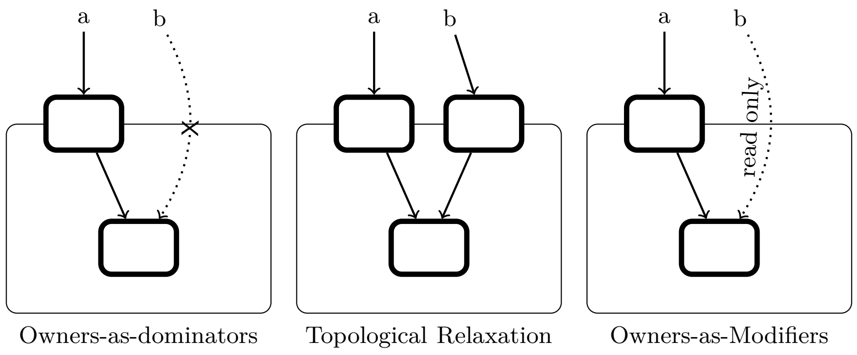

Fig. 1. Various Ownership Disciplines

enclosing object, in addition to its owner's direct representation, its own representation, and any object at the outermost level of nesting. As a consequence of these constraints, program heaps are tree-structured—a object is inside its owner. The owners-as-dominators invariant provides a strong notion of encapsulation that requires that all external accesses to an object's internals must go via its interface, its owner.

The canonical ownership types example is a list whose links are its representation. In an owners-as-dominators system, the links of the list cannot be aliased outside of the list. This protects any class invariants regarding, for example, the ordering of elements in the list.

```
class List[owner|data] {
  Link[this,data] first;

  Iterator[owner,data] iterator() {
    Iterator i = new Iterator();
    i.current = first; // untypeable
    return i;
  }
}

class Link[owner|data] {
  Link[owner,data] next;
  Object[data] data;
}

class Iterator[owner|data] {
  Link[X,data] current; // not possible -- what is X, this or owner ?
}

List[x,y] myList;
Link[?,y] aLink = myList.first; // not possible -- what is '?' ?
```

The topological restriction of the links is due to the type of `first`, namely `Link[`**`this`**`, data]`—**`this`** in the owner (first) position of the type specifies that objects of that type are the representation of the object currently denoted by **`this`**. The owner **`this`** is only visible internally to the object, meaning the appropriate type necessary for referencing the `List` from a variable cannot be formed in some other scope, unless the permission to reference the `List`'s direct representation has been explicitly passed to that scope. The type system prevents such `Link` objects being passed out of the `List` object that owns them.

In most Ownership Types systems, types are parameterised by what can be seen as permissions to reference external objects. The first permission parameter doubles as the owner parameter and is always present in a type. Type compatibility within a given scope requires that the corresponding owner parameters of the two types are the same. Passing between scopes, such as when accessing a field of an object or passing an object as a method parameter to another object, requires that the types involved are transformed to test compatibility. This is done by substituting the owner parameters into the type being translated between scopes. As **`this`** has no meaning to an external object, types with **`this`** cannot be translated. In classical Ownership Types, preventing types of representation values from being valid externally provides a strong notion of protection.

For concreteness, in systems supporting owners-as-dominators, the rule enforced for an object a to validly reference object b is that either

1. a is the owner of b,
2. a and b are siblings, or
3. b is outside a.[2]

The meaning of outside is the converse of the inside relation. The inside relation is the reflexive, transitive closure of the owned-by relation. That is, a is inside b if $a = b$ or if a is owned by b, transitively. (The reflexive part of the definitions of inside and outside is confusing.)

The strength of the owners-as-dominators property is that it provides a simple, clear and strong guarantee. This can be useful when reasoning about various properties of the code, as we discuss later in the chapter. Unfortunately, the invariant makes programming more difficult, as common idioms involving aliasing cannot be expressed.

3.2 Ownership and Subclassing

Ownership typing interacts with subclassing in a relatively straightforward fashion. To enforce the owners-as-dominators invariant in the presence of subtyping, the nesting relation between owners is lifted into the type system, and the nesting assumptions on the owner parameters of a class need to be satisfied for a type to be well-formed. Otherwise, inherited code might make nesting assumptions on owners that are not satisfied in a derived class. Finally, the owner needs to be preserved. That is, the owner of the superclass needs to be the same as that of the subclass [44], and it can never be eliminated via subtyping, since that would be equivalent to an object losing information about its owner, and owners are used to determine who can access an object.

The example below defines a class `Circle` and another class `ColouredCircle` which subclasses the first. `Circle` has two owner parameters—its mandatory owner and `point`, which is the owner of a point object that will act as the circle's centre. The subclass `ColouredCircle` also takes two parameters, **`owner`** and `colour`, and instantiates its superclass' `point` parameter with **`owner`**—expressing the constraint that in coloured circles, the centre point object must be a sibling of the coloured circle object itself.

[2] As pointed out by one of our reviewers, these rules are very similar to Algol's scoping roles for block-structured languages.

```
class Circle[owner|point outside owner] {
  Point[point] centre;
}

class ColouredCircle[owner|colour outside owner]
                    extends Circle[owner,owner] {
  Colour[colour] c;
}
```

When calculating a type's supertype, the extends clause in the class declaration establishes a mapping from parameters in the subclass to parameters in the superclass. Based on this, the supertype of `ColouredCircle[a,b]` is `Circle[a,a]`, which loses information `b` from the type. In Java-like ownership systems such as Joline [42], `Circle` has an implicit extends clause **`extends`** `Object[`**`owner`**`]`, which is a straightforward adaptation of Java. Consequently, the only possible supertype of `Circle[a,a]` is `Object[a]`.

In the example above, the following four owners are accessible when forming the supertype in `ColouredCircle`: **`owner`** and `colour` (from the parameter declaration), **`this`** (denoting the coloured circle's representation) and **`world`**, which is the outermost owner in which all others are nested. Giving the coloured circle the super type `Circle[`**`world,this`**`]` would not be valid for two reasons. Firstly, the owner parameter must always be the same.[3] Secondly, passing **`this`** as a parameter is not valid since it is not nested outside **`owner`**—the condition `colour` **`outside owner`** would be invalidated.

Constraints similar to the ones described above are applicable to other Ownership Types systems.

Regarding the `List` example above, owners-as-dominators excludes many common programmings patterns such as external iterators, since that would require an external object (which is not the list itself) that has a reference to a list's links. To support more programming patterns—and for other reasons—, several systems introduce relaxations of owners-as-dominators: such as temporary, stack-based relaxations (Section 3.3), owners-as-modifiers (Section 3.4) and owners-as-ombudsmen (Section 3.6). Beyond these, Ownership Domains (Section 3.5) avoid imposing a single policy, but instead allow programmers to specify policies for certain data structures or for a whole program. Multiple ownership systems (Section 3.6) do not impose a tree structure on the heap, but it uses an effects system to allow reasoning about the origin or target of a strong update.

3.3 Stack-Based Relaxations of Owners-as-Dominators

Most ownership systems support some kind of relaxation of the topological property for stack-based variables. In Joe$_1$ [44], `myList`'s representation can be typed outside of the list using the `myList` variable as the external name corresponding to the internal name **`this`**, provided the variable is not assignable, following

[3] The only case where this is not true is when *manifest ownership* [39,119] is used, which occurs when all instances of a class will have the same owner. In this case, the class has no parameters, and the superclass's owner will be some fixed owner.

the so-called ‘dot notation’ [35]—these are a form of path-dependent type. This mechanism allows both the creation of representation objects external to the intended owning object, as well as returning temporary references to internal objects. Although this mechanism temporarily relaxes the topological restriction, it could be used safely to implement iterators that could access a list’s links, without allowing the iterator to escape the dynamic scope in which it is defined. In any case, one can view this as a mechanism vs. policy issue. Such owners provide a mechanism for temporarily violating the protection, but it can only be used if the interface of the class exposes methods/fields with `this` in their type.

Joline [42] and Buckley’s thesis [29] take a different approach, and instead support a notion of borrowing implemented through owner-polymorphic methods: a method may be granted permissions to reference any object accessible to its caller for the duration of its execution. (Owner-polymorphic methods were also suggested in Clarke’s thesis [39].) This structured principle allows an object to give out temporary permission to reference its representation when calling a method on another object, with the guarantee that when this method returns, all such references will have been invalidated. This does not, however, allow external initialisation of representation, nor returning references to representation, which is possible in Joe$_1$. Hence, it is not flexible enough to express iterators in any direct fashion.

Joline also introduced the notion of *generational ownership*, which enabled new owners to be introduced for the life-time of a stack frame (called scoped owners). Objects can be created to be owned by these owners, thereby allowing an entire heap whose life-time is tied to a particular stack frame, reminiscent of regions in the region calculus [134]. Objects in such a heap can refer, in principle, to any object in the main heap or any object in the heap associated with a pre-existing stack frame, so long as the appropriate permissions have been passed in. This is depicted in Figure 2.

Boyapati’s SafeJava [20,22] relaxes owners-as-dominators for instances of Java inner classes. In Java, an inner class is always instantiated relative to some enclosing instance to which it has privileged access, that is, access to its private members. To preserve owners-as-dominators, instances of inner classes should belong to the enclosing instance’s representation, however SafeJava allows instead instances of inner classes to be owned by the owner of the enclosing instances. This allows patterns such as iterators to be expressed. From a programming standpoint, this relaxation is always intentional as the inner class must be provided explicitly and it is internal to the class whose representation is to be exposed. Systems for multiple ownership (see Section 3.6) formalise this style of ownership, but address it in a more structured fashion with a clearer semantics.

3.4 Owners-as-Modifiers

The owners-as-modifiers property, first introduced in the Universes type system [105,54], relaxes owners-as-dominators for *read-only references*. In the context of Universes, a read-only reference is a reference which can only be used to read fields and to call *pure methods*. Pure methods may not modify any

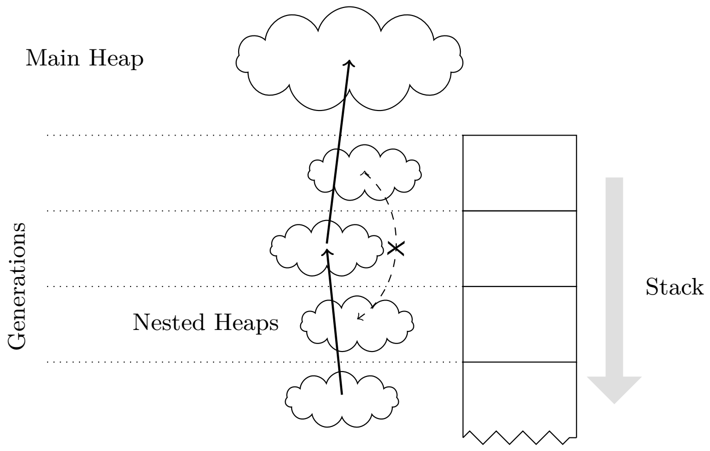

Fig. 2. Generational Ownership. References can refer to earlier generations, but not newer ones.

existing object, including their receiver. The underlying principle is that aliasing is unrestricted, but modifications of an object can only be initiated by its owner. Although the owners-as-modifiers discipline was originally inspired by Flexible Alias Protection, the main driving force behind the design of Universes has been requirements coming from the verification of object-oriented programs. Indeed, Universes have been used extensively to support the verification of object-oriented programs [107,106], and they have been integrated into JML, the Java Modelling Language [56]. Universes have been carefully formalised and proven sound using Isabelle/HOL [85] based on a Featherweight Java-like system extended with field updates.

In systems supporting owners-as-modifiers, the rule enforced for an object a to validly reference object b through a reference r, is that either

1. a is the owner of b,
2. a and b are siblings,
3. b is outside a—permitted in Generic Universe Types [54], but not Universes—, or
4. r is a read-only reference and only pure methods can be called on it.

The following code revisits the `List` example from owners-as-dominators. Here, the keyword **`rep`** has the same meaning as **`this`** above when interpreted as an owner, and **`peer`** is the same as **`owner`**. The keyword **`any`** is new and denotes a read-only reference to an object with unknown owner. The lack of permission parameters requires that the `Link`s of a `List` store read-only references to their data elements—though this can be fixed using generics. The line marked `(***)` shows that leaking references to representation objects is possible, but only via read-only references.

```
class List {
  rep Link first;

  peer Iterator iterator() {
    peer Iterator i = new Iterator();
    i.current = first;
    return i;
  }
}

class Link {
  peer Link next;
  any Object data;
}

class Iterator {
  any Link current;
}

rep List myList;
any Link aLink = myList.first; // ok, aLink is read-only (***)
```

The owners-as-modifiers discipline increases the flexibility of the reference structures that can be expressed and relieves the programmer of the burden of propagating permissions through the code, at the cost of losing modification rights. Mode **any** expresses that the programmer does not care about ownership information. While this loses topological information associated with owner names, this is a design choice, as **any** references are used not to restrict the topology of a program, but to enforce encapsulation—modifications cannot occur through an **any** reference. Generic Universe Types [54] also include a mode **lost** which refers to indicate that information about ownership has been lost in the type system ('don't know'). References with this mode cannot be updated; the presence of **lost** is like an existential type, with a restriction on the operations permitted on such references.

Later work on Universes adds generic types to the underlying language and further separates the mechanisms to specify encapsulation and to restrict the topology of the object graph [54,60]. This separation of concerns allows for a cleaner formalisation and the better reuse of the two mechanisms. A detailed analysis of this system has been performed, resulting in the following characterisation: no modification to an object can occur unless the object's owner appears as the target of a method call on the stack. Similar characterisation theorems have not been presented for other Ownership Types systems.

3.5 Ownership Domains

In an effort to decouple the underlying topological invariant from the language definition, Aldrich and Chambers proposed the notion of Ownership Domains [4] with the purpose of separating the encapsulation policy from the mechanism expressing policies, thereby allowing different aliasing policies for different circumstances.

Ownership Domains is a flexible system in which programmers specify one or more *ownership domains* (which act as the owners) for each object and then explicitly link these together to control the permitted reference structure of a (part of a) program. Objects in a public domain are accessible to everyone which can access the object enclosing the domain—they are considered part of the object's interface. In contrast, objects in a private domain are encapsulated inside the enclosing object. Public domains express containment, private domains express topological restriction. In addition, the links between two domains specifies that objects in one domain can access objects in the other domain. In Ownership Domains, all domains are explicitly named, as this arguably conveys design intent better than an implicit context can. The resulting system is therefore very flexible, and can express both different kinds of invariants than other ownership systems. In particular, Ownership Domains can express more than one private domain and how different parts of an object interact in terms of aliasing.

The following code example defines a linked list in Ownership Domains with support for an iterator inspired by an example in the original Ownership Domains paper [4][4] A list defines a private domain `cells` for the `Cells` and a public domain `iterators`. Data elements in the `List` are **`shared`**, the equivalent of **`world`** above (this is a simplification in this example), and therefore accessible by all. The `Cells`, however, are completely encapsulated in the `List` object, except for references from objects in the public `iterator` domain. This domain will only contain `Iterator` objects, which are only accessible to objects that can refer to the list itself.

```
class List {
  private domain cells;
  public domain iterator;
  link cells -> shared;
  link iterator -> shared;
  link iterator -> cells;

  links Cell first;

  iterator IteratorI iterator() {
    iterator Iterator<cells> i = new Iterator<cells>();
    i.current = first;
    return (iterator IteratorI) i;
  }
}

class Cell {
  shared Object element;
  owner Cell next;
}
```

[4] Class `Link` is renamed `Cell` to avoid confusion with Ownership Domain's keyword **`link`**.

```
interface IteratorI { ... }

class Iterator<what> extends IteratorI {
  what Cell current;
}
```

The inherent flexibility of the system shifts the problem of fitting a program to a given aliasing policy to correctly expressing, or proving, that a program conforms to a certain policy. For example, giving objects in a public domain access to objects in a private domain exposes them via proxy objects in a way similar to the inner classes in SafeJava, discussed above, though independently of the class hierarchy. The constraints assumed/imposed by link declaration in Ownership Domains need to be satisfied when building types; furthermore, they are propagated when subclassing. A consequence of this they will be preserved in the presence of subtyping.

One of the potential problems of Ownership Domains comes from its flexibility, which may make it difficult to understand the consequences of a given collection of annotations. One way around this problem is to use tools to help visualise and hence understand the structure imposed by annotations. In this direction, Abi-Antoun et al. [3] propose SCHOLIA, a tool for extracting conservative approximations of runtime object graphs from static ownership domains annotations. Such graphs may also help identifying deviations from the desired encapsulation policy.

To further increase the flexibility of Ownership Domains, Schäfer and Poetzsch-Heffter developed Simple Loose Ownership Domains [124]. This model keeps the public and private domains of the Ownership Domains model, though it hard codes a single private and single public domain per object, whereas Ownership Domains allows multiple public and multiple private domains per object. In addition, Simple Loose Ownership Domains omit link and domain declarations to reduce the syntactic overhead at the cost of a loss of the ability to express fine structure. Each object has a boundary domain which stores objects that are both publicly accessible and can access private objects. In addition, their model supports loose domains which allow one to abstract from the precise domain to which an object belongs, though for soundness reasons field update and method call are prohibited on loose domains. Their system enforces a property referred to as *boundary-as-dominator*, which means that the only access paths to an objects representation are via objects that it advertises as boundary objects, namely, those in boundary domains.

Although not the same as Ownership Domains, Lu and Potter [96] adopt the ‘separate mechanism from policy’ philosophy and explore a type system that separates the ownership assignment from the restrictions imposed on owners in, for example, the original Ownership Types system. Their model also allows ownership to vary, which can be seen as a form of ownership transfer. The type system specifies not only who owns an object but also who can reference it.

Their system is similar in spirit to ownership domains, but the underlying mechanism is more lightweight in Lu and Potter's system. This is also one of the few type systems that permit owner variance, using lightweight, programmer-specified variance annotations, which increases the expressiveness of the language.

3.6 Multiple Ownership and Owners-as-Ombudsmen

Several researchers have identified problems with the strong topological requirement imposed by ownership types, namely that having single owners for objects requires that the ownership relation embedded in the heap is organised into a tree-shape. But many programs and idioms do not fit into a tree structure, such as the iterator example discussed above.

To express design patterns where multiple objects interact and share ownership of objects, Cameron et al. [33], proposed a system of multiple ownership, called MOJO, wherein the ownership relation instead forms a DAG. MOJO does not impose any particular topology on a program's heap, rather, it relies on an elaborate effect system to statically capture interference using ownership information, thereby imposing an encapsulation discipline in spite of the looser topology. Following up on MOJO, Li, Cameron and Noble proposed the Mojojojo system [91]. Mojojojo simplifies and generalises MOJO; for example, it can express that an object lives in the intersection or the union of two objects' representations, which is useful for expressing sharing constraints while still preserving some locality.

Östlund and Wrigstad's *owners-as-ombudsmen* proposal [113] relaxes the owners-as-dominators property by allowing multiple objects to define a shared *aggregate owner*. An aggregate owner can have one or more *bridge objects* between the representation and the external objects. This allows components to express the topology underlying a restricted common state with multiple entry points to it. With owners-as-ombudsmen, the dominator for an object inside an aggregate is the dominator of the bridge objects defining the aggregate.

In systems supporting owners-as-ombudsmen, the rule enforced for an object a to validly reference object b is that either

1. a is the owner of b,
2. a and b are siblings,
3. b is outside a, or
4. a is owned by the aggregate owner b.

The owners-as-ombudsmen topological invariant can be understood as a simplification of Mojojojo, without requiring the effects system.

Using owners-as-ombudsmen, iterators can be expressed in a way that makes the `List`'s `Links` part of an aggregate defined by the `List` and its `Iterator` objects.

```
class List[owner|data] {
  Link[aggregate,data] first;

  Iterator[bridge,data] iterator() {
    Iterator[bridge,data] i = new Iterator[bridge,data]();
    i.current = first;
    return i;
  }
}

class Link[owner|data] {
  Link[owner,data] next;
  Object[data] data;
}

class Iterator[owner|data] {
  Link[this,data] current;
}

List[x,y] myList;
Iterator[x,y] iter = myList.iterator(); // note bridge -> owner
```

The type `Iterator[`**`bridge`**`,owner]` captures the fact that the `i` variable points to another bridge object of the shared `aggregate`. When an external object calls the `iterator` method, it will see a type that has the same owner as the list itself, since **`bridge`**, like **`this`**, is an owner which is not visible externally. As a consequence, the links are writeable by sibling objects.

Another approach that produces the effect of multiple object owners is Tribal Ownership [34]. This was proposed by Cameron, Noble and Wrigstad and relies on earlier work by Clarke et al. on the virtual class calculus Tribe [41]. In Tribal Ownership, ownership nesting is reflected in the nesting of virtual classes and each object has an **`out`** reference to its enclosing object, equal to being able to name one's owner. Tribal Ownership allows different prescriptive ownership policies to be plugged into the system which gives rise to different levels of protection. Tribal Ownership furthermore allows a novel owners-as-local-dominators policy, which allows owners-as-dominators to be enforced in local subheaps of a program, as opposed to having one single system which imposes owners-as-dominators on the entire heap.

3.7 Confined Types

Confined Types [136,137] are a lightweight approach to enforcing the confinement of objects. Very few annotations are required to express the desired confinement discipline, at the cost of some expressiveness compared to other ownership systems. Confined Types, in their original form, enforce the following informal soundness condition:

An object of confined type is encapsulated within its defining scope [148].

This is enforced using a small set of annotations and a statically checkable set of rules. Firstly, classes may be annotated as **`confined`** to indicate that their instances are confined types and cannot be accessed outside their defining package. Secondly, methods that can safely be inherited by confined types must be marked as **`anonymous`**—anonymous methods cannot export the **`this`** reference. Finally, there are a collection of statically checkable rules such as the following (quoting [148]):

C1. A confined type must not appear in the type of a public (or protected) field or the return type of a public (or protected) method.

C2. A confined type must not be public.

C3. Methods invoked on an expression of confined type must either be defined in a confined class or be anonymous methods.

C4. Subtypes of a confined type must be confined.

C5. Confined types can be widened only to other confined types.

C6. Overriding must preserve anonymity of methods.

A1. The **`this`** reference is used only to select fields and as the receiver in the invocation of other anonymous methods.

The first six rules ensure that instances of some confined type do not escape the scope by ensuring that it does not appear in the interface of a public class (C1), that the class itself is not public (C2), that confined value do not leak via untrusted methods (C3) or by forgetting that the type is confined (C5). The rules (C4) and (C6) ensure that the property of being confined and being an anonymous method is preserved via subclassing. The last rule (A1) ensures that anonymous methods do not leak the **`this`** reference. If these rules are observed, the Java compiler does the remainder of the checks, even when compiled against code is not aware of the Confined Types discipline.

In the following code sample, all instances of class `Link` are confined to the package `listpackage`:

```
package listpackage;

class List {
  Link first;

  Iterator iterator() {
    Iterator i = new Iterator();
    i.current = first;
    return i;
  }
}

confined class Link {
  Link next;
  Object data;
}
```

```
class Iterator {
  Link current;
}
```

Confined Types are related to ownership types, but they differ in two significant ways. Firstly, confined types attempts to reduce the syntactic overhead imposed by the type system by relying on package- or class-level annotations or defaults, thus avoiding the annotation of types. The second difference is the degree of confinement provided. In earlier systems of Confined Types[136,137], the degree of confinement was at the package level, meaning that confined objects could only be referenced by other objects within the same package. Later systems achieved object level confinement [9,145,133], though without the same degree of flexibility as Ownership Types—types parameterised by the owner of their members cannot be expressed. The original Confined Types system [136,137] was presented as a collection of informal rules. These were latter formalised and proven to be sound [147,148]. An inference tool was also developed for Confined Types to make it easier to apply them with existing code bases [69].

The original motivation of Confined Types was to address some security properties that could not be expressed in Java's type system. This security property was extended further and applied in the context of Enterprise Java Beans to ensure that beans do not escape their defining context without being appropriately wrapped [47]. Applications of Confined Types to memory management [9,145,133] are surveyed in more detail in Section 6.3. Both Reflexes [129] and SteamFlex [130] apply notions of implicit Confined Types in the context of high performance stream processing applications. In all of these systems, the underlying sets of rules differ—each is tailored to the specific application domain.

4 Extensions

Ownership Types systems have been extended in a number of dimensions beyond the kind of policy they enforce. Ownership has been combined with generics and computational effects systems. More dynamic notions of ownership have been explored, including systems supporting a notion of ownership transfer. Ownership has also been explored beyond the mainstream object-oriented paradigm. The remainder of this section explores these topics.

4.1 Ownership and Generics

Modern programming languages support generic classes and bounded parametric-polymorphic type systems. A natural question is how ownership interacts with these mechanisms, especially considering that both mechanisms introduce a kind of parameterisation into classes.

The original Flexible Alias Protection proposal [112] was phrased in terms of a language with generics. Generic parameters served as a vehicle for delivering aliasing modes into a class in the sense that the alias modes annotated generic parameters, and only generic parameters, to avoid too much syntactic overhead. Such an approach does not allow ownership parameterisation isolated from type parameterisation.

In Clarke's PhD thesis [39] Ownership Types are encoded in terms of Abadi and Cardelli's object calculus [1]. Clarke also adapted Abadi and Cardelli's encoding of classes to include ownership in the obvious manner: as genericity is achieved using type parameters, ownership polymorphism is achieved using owner parameters. Both can also be constrained by the appropriate kinds of bounds.

SafeJava [20] offered both type parameters and ownership parameters in independent syntactic categories, but this can lead to significant annotation overhead. The language underlying Ownership Domains also included both type and ownership parameters in a single parameter space [4]. Constraints on the owners of type parameters could optionally be specified to help define the relationship between various ownership domains. Generic Universes [53] also separate type and ownership parameters. Only ownership parameters were, however, included in their formalisation. It is arguable that there are cases when it is useful to have ownership parameters independently of type parameters, and that it would be unnatural to introduce a type parameter just to pass around an ownership parameter.

The idea of piggybacking ownership (and other) information onto generic parameters, instead of treating them as orthogonal, has been explored extensively by Potanin and his coauthors [119,120,150]. With a suitable choice of defaulting mechanisms, the approach reduces the annotation overhead and the conceptual burden, as classes take only one kind of parameter. For example, using piggybacking, the class declaration

```
class List[owner|data outside owner,Data extends Object[data]] { ... }
```

can be replaced by

```
class List[owner|Data extends Object[data]] { ... }
```

Here the owner `data` is not explicitly declared, but it presence and the constraints on it can be inferred from the context. Similarly, when forming an element of this type, the implicit `data` parameter need not be specified. The improvement this approach offers is much greater when the class has more parameters. Potanin et al. applied this technique to confinement [119], ownership [120], and ownership and immutability combined [150].

Jo∃ [32,30] adds existential types on top of a generic Ownership Type system. Ownership information is passed as additional type parameters and existential quantification allows owners to vary. The main advantage of existential quantification is to allow more precise reasoning about unknown owners—rather than marking them with a '?', existential types can be used, thereby naming the owners and being more explicit about the relationship between them: compare types `C[?|?]` with `∃o.C[o|o]`: the latter expresses a relationship between the two owner parameters, even though they are unknown. A variant Jo$\exists_{deep}$ enforces the owners-as-dominators policy.

Dietl, Drosspoloulou and Müller [53,60] extended Universes to include generics. This was the first type system to combine the owners-as-modifiers discipline with type genericity. Their approach also aims for a seamless integration of genericity with the ownership mechanisms and enables the separation of the specification of the topology from the encapsulation constraints, which opens the door for more flexible systems to be expressed [54].

4.2 Ownership and Effects

Computational effects systems, such as those expressing abstractly the possible field reads and writes a method can perform, become more when combined with Ownership Types. Clarke and Drossopoulou [44] demonstrated that combining an Ownership Types system enforcing the owners-as-dominators policy with an effects system offers strong guarantees, not only about the object on which the method is called but on whole chunks of the heap. This system also included a notion of sub-effecting that exploited the hierarchical structure of the ownership tree. Taking a different view on ownership and effects, Yu, Potter and Xue [98] present an alternative to Ownership Types based on effect encapsulation instead of restrictions to the reference topology. References may leak out of their defining scope, but what can be done with those references is limited using an effects system. This system can also be considered as an owners-as-modifiers system, due to the constraints imposed on leaked references.

The combination of ownership and effects is particularly important for reasoning about concurrent systems [82,83,84], and for guaranteeing race and deadlock freedom [21,20,24,51]. See Section 6.1 and Bocchino's chapter [19] for more details. Boyapati et al.'s application of Ownership Types to object upgrades also relies on effects to achieve modularity [23]. Yu and Potter use Ownership Types and effects to reason about object invariants based on the notion of validity effects that capture the objects that may be invalidated by some code block [97]. The multiple ownership type system uses effects to reason about when different owners are guaranteed to be disjoint, even though the ownership relation can be DAG shaped [33]. Finally, Clifton et al.'s MAO combines ownership and effects to reason about aspect-oriented programs [48].

The related notions of readonly references and immutability limit computational effects without requiring the tracking of effects by building permissions (such as whether a field write is permitted) into types. As already mentioned, Universes [105,54] have a notion of readonly reference built in. Östlund et al. [114] present a system combining ownership, uniqueness and immutability to obtain more powerful invariants than would be possible without them. For instance, the system allows the staged initialisation of immutable objects, meaning that an object can be initialised, hence mutated, in multiple places before eventually becoming immutable. Ownership Immutability Generic Java (OIGJ) [150] extends Featherweight Generic Ownership [120] to capture both ownership and immutability within a single type system, leveraging off Generic Java's type system, without introducing new syntactic categories to, capture notions of ownership and immutability. Immutability is considered in more detail in Potanin et al.'s chapter [121].

4.3 Dynamic Ownership

In early Ownership Types systems, checking is performed purely statically. While this provides strong guarantees, it has expressiveness limitations that make exploratory programming difficult. Various degrees of support for dynamic ownership have been explored, including run-time ownership information to support downcasts and Gradual Ownership typing, all the way to fully dynamic ownership where all checking occurs at run-time. The information required at run-time can be simply the owner of each object, but it generally includes the values of all owner parameters and the run-time nesting relationship between objects.

Systems supporting dynamic type casts include SafeJava [20], Generic Universe Types [53], and Gradual Ownership Types [127]. In terms of the amount of checking performed, approaches supporting downcast perform checks only when an explicit downcast is made, whereas Sergey and Clarke's Gradual Ownership typing approach also performs boundary checks when objects are passed between different objects. As an alternative approach to downcasts, Wrigstad and Clarke [142] present a lightweight approach to run-time downcasts which relies on existential types. Downcasting from a well-formed type `C[a]` to another well-formed type `D[a,b]` can be compiled as a regular downcast from `C` to `D`, ignoring ownership. If the cast succeeds at run-time, the additional owner parameter introduced by the downcast must exist, and its relation to owner `a` can be inferred from the declaration of the class `D`. This allows the introduction of the owner `b` as an existential owner parameter visible on the stack in a branch where the downcast was successful, without any need for run-time owner representation.

Dynamic ownership delays the checking of the properties expected by ownership types systems until run-time. A preliminary experiment of this idea was performed in the context of a prototype-based programming language [111]. This work was adapted to a class-based setting by Gordon and Noble, who introduced the scripting language ConstrainedJava [67]. The ownership structure is represented using an owner pointer in every object. Operations are provided to make use of and change these owner pointers. The semantics of the language relies on a message-passing protocol with a specific kind of monitoring. Messages are classified into several categories based on their relative positions of the message sender and receiver in the ownership tree. "Bad" messages are detected using run-time monitoring.

Leino and Müller [90] make use of dynamic ownership in the context of Spec# to control which parts of the heap class invariants may depend on. In contrast to most other ownership systems, the ownership relations of their system are conditions that need not always hold. Invariants may, for example, be temporarily broken during ownership transfer, as this is not an atomic operation, and involves passing the reference and changing the owner field of the moved object.

4.4 Ownership Transfer

One restriction common to early ownership systems is that the owner of an object must be set upon creation and then fixed for the lifetime of the object. Several attempts at removing this restriction have been presented over the years, thereby allowing the transfer of ownership.

Older systems, AliasJava [5], SafeJava [20] and Flexible Alias Protection [112] provide limited notions of uniqueness, corresponding to a reference that has not been assigned to a field or is stored in just one field (that is, no aliases). While this appears to be a perfectly reasonable notion of uniqueness, it fails to exploit the structure of the heap given by the ownership hierarchy. It also suffers from an abstraction problem, identified by Clarke and Wrigstad [42], namely that changes to the internal implementation of a class modifies the behaviour of code using unique references, for instance, an internal change to a method implementation could steal the unique reference upon which the method is called. Such changes need to be reported in the interface of the class, and this change tends to propagate through to client code.

Clarke and Wrigstad's language Joline offers a novel approach uniqueness called External Uniqueness [42,141], exploiting the nesting information provided by Ownership Types, in such a way that the above mentioned abstraction problem does not arise. Ownership Types can identify aggregate boundaries and safely allow aliasing between objects within those boundaries. The approach to uniqueness taken in External Uniqueness is that an externally unique reference is the only reference to an object from outside that object; the internal aliases to an object are permitted and can be ignored in the definition of uniqueness. Thus unique references refer to aggregates not just individual objects. The property enforced by External Uniqueness is called *owners-as-dominating-edges*, which means that all paths to an object accessible through a unique reference must include that reference. This is illustrated in Figure 3. Uniqueness is preserved using a destructive read operation. To preserve owners-as-dominators, however, transfer may only go inwards in the ownership hierarchy. As owners-as-dominators requires that all owner parameters are outside the owner of an object, allowing transfer of objects outwards in the hierarchy would result in a violation of this invariant.

In his Alias Burying proposal, Boyland [26] showed that destructive reads are not necessary for the preservation of uniqueness. Instead, it is sufficient to ensure that any aliases to a unique value are destroyed, for example, when a method exits, and thus preserve uniqueness. Checking that this is the case requires sophisticated static analysis.

Banerjee and Naumann elaborate on sufficient conditions for confinement and object transfer between owners [14,13]. Their objective was to provide a confinement policy similar to other ownership proposals, but without requiring ownership annotations. In order to support transfer, the system builds on a notion of separation similar to External Uniqueness, but which also requires that an aggregate does not have outwards pointing references into the source's representation when transferred to a new owner. This notion was later also adopted Haller and Odersky [71], described below.

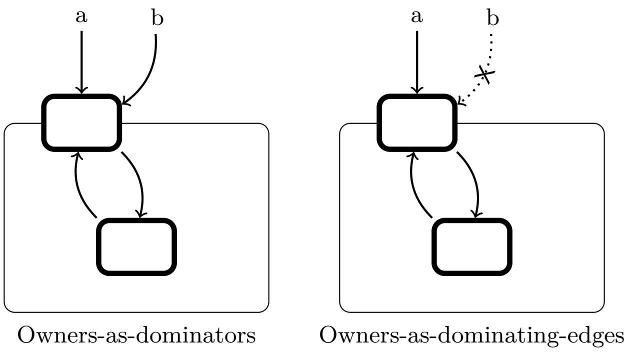

Fig. 3. Owners-as-dominators and External Uniqueness

UTT [108] is Universe Types extended with transfer for externally unique aggregates. In Universes the situation is slightly less involved than in Joline, however, because the types carry less ownership information. The any owner modifier can point inwards, but since it does not convey any ownership information, such references are not an issue for preserving the underlying invariants. Further, Universes does not have owner parameters on classes, and so the restriction on inwards-only transfer is not required. An externally unique aggregate belongs to a specific region, or *cluster*, as they are called in UTT. In place of a destructive read operation, UTT instead employs **`release`** and **`capture`** statements to perform the move. The **`release`** statement will make unusable any external references to the aggregate (similar to Boyland's Alias Burying), stripping the type of information of which cluster it belongs to. The released object is simply free. The **`capture`** statement performs the actual move by assigning a new type specifying the cluster to which the aggregate now belongs.

Haller and Odersky's Capabilities for Uniqueness and Borrowing [71] use a notion similar to that of Banerjee and Naumann, called *separate uniqueness*. Separate uniqueness restricts external uniqueness so that there can be no outgoing pointers from inside a unique aggregate to the outside. This additional restriction helps guarantee race freedom in a concurrent message-passing setting, such as the Scala Actors Framework it is designed to work with. Separate uniqueness employs a capability system instead of ownership as the foundation for keeping track of uniqueness. Interestingly, a swap operator is used to preserve separate uniqueness, much in the spirit of Harms and Weider's 'copy and swap' proposal [72]. Separate uniqueness is maintained by well-formed construction, wherein a separately unique aggregate may be built out of other separately unique objects.

Object Teams also provide a notion of ownership transfer [73]. The mechanism updates all the dynamic information capturing the ownership structure, namely, the roles each object plays, but the mechanism offers no static security.

Anderson et al. [8] apply notions of ownership and ownership transfer in the context of C to describe data structure sharing strategies in multi-threaded programs.

Ownership transfer combines notions of linear (or uniqueness typing) and non-linear typing in the one system. In a non-object-oriented setting, Fahndrich and DeLine's Adoption and Focus approach [63] combines the benefits of linear types with the flexibility of non-linear types in order to enforce software protocols. The model is close to the notions imposed by ownership, as the reference structure considered is hierarchical—linearly typed objects containing within linearly type objects cannot be accessed directly from 'outside'. References start out having linear type, so that the interaction via them can be precisely tracked. However, it is impractical programming with only linear types. To get around this, the adoption operation allows a linear type to be converted to a non-linear type within the scope of another expression. In order to go the other way, the focus construct provides a temporary linear view on an non-linear type, by ensuring that no change made via other aliases. This is achieved by revoking the capability corresponding to the non-linear type. Adoption and Focus was latter generalised as nesting/carving in Boyland's Fractional Permissions [27,28].

4.5 Other Extensions and Variations

A number of alternative extensions and variations of Ownership Types that do not so easily fit into the categorisation above have been developed.

Lu and Potter [95] present a programmer-specified type system for describing reachability constraints in an object graph. The core restriction made by the system is that any cyclic references structures are constrained so that all objects share a common owner.

Ownership Types have been applied to aspect-oriented programming to simplify the task of reasoning about advice [48]. In this setting, Clifton et al. introduce *concern domains* which store objects related to particular concerns (in the sense of separation of concerns). These are used to reason about which parts of data structures are modified by which advice. The underlying type system is based on a shallow ownership-and-effects system.

The interaction-based object-oriented language Classages [94] uses a variant of Ownership Types is used to ensure that certain objects remain encapsulated within components (called classages), whereas other objects can be passed around between components.

Pedigree Types [144] use a relative addressing scheme to traverse the ownership tree, rather than the parameterised approach of Ownership Types. The general form of owner is given by the grammar $\texttt{parent}^k.\texttt{child}^z$, where $k \geq 0$ and $z \in 0, 1$. This can capture all owners accessible in the owners-as-dominators model—**`parent`** traverses up the tree, and **`child`** moves, in effect, to a sibling, except for when **`child`** appears alone, which corresponds to selecting the current object's representation.

Ownership Types have been considered in the context of an object-oriented programming language with relationships [92]. The work covers the problems encountered when trying to combine the two systems, but does not present any solutions to these problems.

Although not strictly following the tradition of Ownership Typing, X10 [37] includes a notion of place types which is similar to Ownership Types in that types partition the space of objects. Places are used to express locality and thereby facilitate better distribution of data across the memory hierarchy of a multicore processor.

5 Ownership Inference

Ownership Types systems generally require a significant amount of annotations to express the types, but this can be burdensome for the programmer. What makes matters worse is that library code also needs to be annotated to work effectively with Ownership Types. Addressing this problem leads naturally to the question of Ownership Type inference. Unfortunately, matters are not so simple. Unlike traditional type systems, ownership annotations are mostly design-driven: it is up to the programmer to decide whether some object should be owned by **`this`** or by **`world`**. Many Ownership Type systems admit a trivial type assignment, for example, by setting all objects to be owned by **`world`**. Consequently, even elaborate approaches to type qualifier inference [38,68] are ineffective, as they infer any solution that satisfies the constraints, but cannot give a best solution.

In this section, we provide a survey of approaches for ownership inference. Two approaches are considered: dynamic inference and static inference.

5.1 Dynamic Inference

Dynamic ownership inference uses snapshots of the run-time object graph to determine an approximation of the ownership structure of the system—these snapshots may involve continual monitoring, in effect taking a snapshot every time the heap changes. The idea is that this information can then be used to help determine a valid ownership typing.

The first work on the dynamic inference of Ownership Types is Wren's master's thesis [140]. The essence of his approach is to run programs with a profiler that keeps track of all heap snapshots, collecting full information about the topology of the heap at any moment. All heap snapshots are then merged and the resulting graph is analysed in order to infer dominance relations between objects. The work provides a graph-theoretical foundation for run-time inference, including a description of the most precise program heap topology with respect to the owners-as-dominators invariant. On the negative side, the dynamically-determined ownership information cannot be mapped directly to types. To remedy this, the author formulates the system of equations to assign annotations to particular object allocation sites.

Dietl and Müller present results on runtime Universe Type inference [57]. As Universe Types require a comparatively lower annotation overhead than Ownership Types, mapping dynamic inference results to static annotations is easier than for the system Wren considered. The inference algorithm is, however, quite in the spirit of Wren's: first a combined representation of the object store is built; then its dominator tree is constructed; finally, conflicts between the information obtained by analyzing the inferred dominator tree and the actual constraints of the type system are resolved by flattening the dominance trees via a procedure referred to as harmonization. The resulting annotations deliver a correct typing of the program with respect to the target type system.

5.2 Static Inference

One of the first attempts to provide Ownership Type inference was by Aldrich et al. [5]. In their system, the programmer needed only to provide a small amount of annotations to indicate the intent that some parts of the program be protected, and the rest of alias annotations were inferred. The approach was not entirely satisfactory, because a large number of parameters were inferred in many cases.

Moelius and Souter [104] employ a variation of an escape analysis [18] to infer ownership annotations. Their algorithm allows borrowed references to be returned from methods and assigned to object fields. No assumptions on ownership parameterisation are made, consequently the algorithm can also result in a large number of parameters, the same problem that Aldrich et al.'s [5] inference algorithm suffered from.

For the same inference problem, Milanova and Liu [101] employ an Andersen-style points-to analysis [7] as part of a static algorithm to infer ownership and universe annotations according to two different ownership protocols: owners-as-dominators and owners-as-modifiers. Both analyses are based on a context-insensitive points-to analysis, therefore they do not distinguish between different allocation and call sites. However, thanks to some Java-related heuristics, their technique handles some idiomatic cases, and good precision is thereby obtained.

Later, Milanova and Vitek presented a static inference algorithm for ownership annotations for the owners-as-dominators invariant based on a static dominance inference algorithm [102]. The algorithm computes approximations of the object graphs using an enhanced global context-insensitive points-to analysis. The candidate ownership annotations are computed based on an approximated dominance tree, built using a variation of *must-point-to* information [52,81]. The approach does not provide any guarantee that the inferred annotations comply with the original Ownership Types system. Naturally, as with any global analysis approach, the issue of scalability is a concern. In subsequent work employing the dominance inference algorithm, Huang and Milanova use the original type checker to verify the correctness of the inferred ownership annotations [80].

Based on the boundary-as-dominator model, where access is permitted either via the owner or other boundary objects, Poetzsch-Heffter et al. [117] present an inference technique that requires that the programmer only annotate the interface types of components and the remaining ownership information is automatically inferred using a constraint-based algorithm. Due to the lack of parameters in the underlying model, this approach delivers reasonable results and represents a good compromise to the inference problem.

Some static analysis-based approaches fail to deliver type annotations directly, but instead extract topological properties similar to those ensured by Ownership Type systems. This is problematic because it make it difficult to view the topological properties in terms of code, and thereby are difficult to reason about. For instance, Geilmann and Poetzsch-Heffter [66] developed a modular abstract interpretation-based analysis to check simple (*i.e.,* non-hierarchical) confinement properties in Java-like programs. This work employs a *box model* [117] instead of dominator trees. The approach is targeted to substitute modular type-checking by modular static analysis, requiring a significantly smaller amount of annotations: only class declarations and object allocation sites need to be annotated. The analysis then takes the implementation of a class, considered as an encapsulated box, and executes it together with its most-general client. The most-general client is an abstraction of all possible clients that is used to create all possible traces through the box. If execution succeeds, the box never exposes any confined object, irrespective of the program that uses the box. The approach is based on formulating ownership as a semantic property of the program and the subsequent construction of an abstraction of the abstract semantics in the style of Cousot and Cousot [49,50].

A general variation of a points-to analysis-based algorithm to infer ownership and uniqueness is presented by Ma and Foster [100]. The algorithm combines constraint-based intraprocedural and interprocedural analyses. The collected information about encapsulation properties is not however mapped to a type system.

Another approach is to generate and solve typing constraints and allow the user to tune the solution. Dietl et al. [55] presented such a static analysis to infer Universe Types according to the user-specified intentions declared with annotations. The first part of the technique is responsible for the generation of equations, based on the program semantics and the rules of the original type system. Constraints of the Universe Type system are encoded as a boolean satisfiability problem. After the constraints have been generated and solved, the second part of the approach is to tune the result of the inference: programmers can indicate a preference for certain typings by adjusting the heuristics or by supplying partial annotations for the program. Dietl et al. empirically demonstrate that the NP-completeness of constraint solving does not result in a significant overhead on real-world programs, compared with other static approaches [80,102,104].

The two lines of research towards Ownership Type inference via points-to analysis and via constraint solving were unified in the work of Huang et al. [79]. The resulted framework implements checking and inference for two systems: Universe Types and Ownership Types. As in the prior work [55], the programmer can influence the inference by adding partial annotations to the program. The algorithms work with a programmer-supplied metric specifying the best typing, which the type inference algorithm attempts to maximise. The underlying analysis is implemented as a Kleene iteration of a monotonic transfer function, based on the program's small-step collecting semantics. The user-provided annotations are taken into account, whereas missing ones are initialised with the bottom element of the appropriate lattice.

An incremental type analysis has also been developed for Confined Types and integrated into the Eclipse platform [62]. The fact that it is incremental means that it provides immediate feedback to programmers as they add their annotations.

6 Applications of Ownership

Ownership Types and related systems have seen many applications. This section surveys their application in concurrency control, verification, memory management, security, object upgrading, and software visualisation and understanding.

6.1 Ownership for Concurrency Control

One application domain where ownership has seen a variety of applications is concurrency. We expect that these approaches merely scratch the surface of what is becoming an increasingly important problem area.

Parameterised Race Free Java (PRFJ) [24] is an ownership-based type system for guaranteeing race-freedom in concurrent Java programs. The discipline PRFJ enforces is called *owners-as-locks*, and is in many ways similar to Flanagan and Abadi's Types for Safe Locking [64], with ownership and parameterisation included for greater flexibility. Each object is associated with a lock, and the encapsulation provided by ownership allows protection of an entire aggregate by acquiring a single lock, namely that of the owner of the aggregate. Further, method annotations reveal what locks must be acquired by the callee prior to method invocation. PRFJ also has thread local variables, annotated with the special owner `thisThread`. This owner may not occur in the type of a field, but only on local variables. Essentially this means that values owned by `thisThread` cannot be shared between threads, and are thus thread local. PRFJ was subsequently extended to ensure deadlock-freedom [21]. Based on programmer-supplied annotations, a partial order on locks could be established. The type system then rejects any programs that do not adhere to the partial order when acquiring locks, again in a similar fashion to Flanagan and Abadi [64]. Permandla et al. [116] continue work in this direction, by designing a similar type system for Java bytecode. The type system enforces race- and deadlock freedom of precompiled files at load time.

Cunningham, Drossopoulou and Eisenbach [51] use the Universes ownership model as the basis for a race-free type system. The system shares many similarities with PRFJ, but it uses a simpler type system that aims to be much more user friendly. To cope with some of the reduced expressiveness, the system uses effects to deal with references to domains not identifiable by immutable paths. A number of extensions were presented that can distinguish between read/writes, prevent deadlocks, verify atomicity, and allow locks to be taken at the granularity of single objects. The authors conjecture that adding genericity to the type system will increase expressiveness.

For the purpose of supporting large numbers of concurrent object-oriented actors, Srinivasan's and Mycroft's [131] implement cooperatively-scheduled green threads on the JVM in their Kilim language. To achieve isolation of actors without incurring a huge copying overhead, they require that messages between actors be tree shaped by using only unique references. These can then trivially be passed between actors at a close-to zero, constant-time cost. In contrast, Clarke et al.'s Joelle language [43] employs ownership types to allow zero-overhead confinement of active objects, each with a single thread. By employing ownership, messages can have complicated graph-like or circular structure and not be limited to simple trees. In Joelle, external uniqueness [42] suffices for efficient object transfer, and where external uniqueness cannot be established, ownership information can be used to perform "sheep cloning" [111], which calculates the minimal safe clone statically. Joelle furthermore supports the safe sharing of immutable subgraphs of otherwise mutable data, similar to the **`arg`** mode of Flexible Alias Protection [112].

The CoBox (concurrent box) [125] concurrency model unifies the active object model with structured heaps in a similar manner to Minimal Ownership [43]. A key difference is that in the CoBox model multiple objects play the role of the active object. This is realised by associating a single lock with each cobox to ensure that at most one of the objects within each cobox has a thread of control at a time. A Java-based implementation of the CoBox model exists [126], and CoBoxes were incorporated into the ABS programming language [40], where they are called *cogs* (concurrent object groups).

Loci [143] is a type system for thread local data based on the notion of *owners-as-threads*. Using a simple ownership system based on a conceptual division of the heap into a shared area and a private area per thread, objects can be determined to belong to either the shared heap or be thread local, thus belonging to one of the threads. The type system ensures that shared and thread local data are never confused, thus preventing accidental sharing. Thread local objects can be accessed without synchronisation. A similar model is found in the older system Guava [10], except that Guava's rules are presented informally, whereas Loci is completely formalised and proven correct. Guava is a Java dialect that guarantees that shared data is accessed only through synchronised methods. In Guava there are no synchronised methods (in the Java sense). Instead, classes are split into two kinds: monitor classes, where all access are fully synchronised, and ordinary classes, whose instances can only be shared within a single thread, and are thus thread local.

As mentioned in Section 4.4, Haller and Odersky [71] present an Ownership Types system for expressing uniqueness and borrowing to be used with Scala's actor model. The type system ensures that objects can be safely passed between actors without leading to race conditions, thereby avoiding the cost of object cloning.

In a series of papers, Bocchino and his coauthors investigate the use of type and effect systems similar to Ownership Types with effects to enforce a notion of Deterministic Parallelism [82,83,84]. As these efforts are surveyed in another chapter in this volume [19], we keep our discussion brief. After introducing the initial system [83], subsequent work [82] presented a type system for writing user code that will operate properly when used with parallel object-oriented frameworks such as Map-Reduce. The approach is similar to Clarke and Drossopoulou's ownership plus effects system [44], albeit tailored to the demands of a particular application domain. A subsequent paper [84] extends the system to permit safe nondeterminism using special blocks declared by programmers, thereby providing mechanisms encapsulating and controlling the nondeterminism.

Task Types [88] are another approach for preserving atomicity in multithreaded programs. The underlying structures are similar to those enforced by Ownership Types, except that the type system helps express explicit sharing in a more explicit fashion.

6.2 Ownership for Verification

Ownership has played a very solid role in verification, helping deal with issues of *framing*, knowing which properties are affected by a given code block, *invariants*, knowing which properties can be relied on, and *locking*, knowing when a lock needs to be obtained to avoid data races. Ownership has been used both via typing or by encoding the desired invariants into specifications. Notions of ownership have been incorporated into specification languages JML [106], and Spec# [90].

Banerjee and Naumann use ownership to show representation independence properties of classes [11,16,15], which enables one implementation of a class to be replaced by another. Ownership is used to indicate which classes are hidden behind the abstraction boundary. Their work has addressed this problem for increasingly sophisticated program models and relaxed restrictions.

Much more could be said here. Instead duplicating other excellent work on the topic, we invite the reader to consult other the chapters of this volume that discuss the role of ownership in verification [17,58].

6.3 Ownership for Memory Management

The fact that Ownership Types could be applied to memory management was identified early on [122]. It took some time before anyone explored the idea, and all work to date has been done in terms of RTSJ or related systems.

Before continuing, we first present a little background. The Real-time Specification for Java (RTSJ) memory model includes various regions of memory: immortal memory, heap memory, and numerous programmer specified scoped memories. Immortal memory is for objects that remain for the entire application. Heap memory is garbage collected. Scoped memories are allocated and deallocated in a stack-like fashion based on the order in which threads 'enter them' to allocate objects within them. Without going into too much detail, objects in one scoped memory can refer to objects in another, if the lifetime of the former exceeds that of the latter, to avoid memory leaks. RTSJ checks dynamically that the scoped memories are used correctly.

Boyapati et al. [25] were first to explore the application of Ownership Types to memory management in any detail. Their work combines Region Types and Ownership Types in a unified framework to statically enforce object encapsulation and region-based memory management. The memory model underlying the type system is compatible with RTSJ's memory model. Additions beyond the basic ownership machinery include subregions within shared regions to allow long-lived threads to share objects without using the heap and without memory leaks, typed portal fields for enabling inter-thread communication, and thread local regions.

ScopeJ [146] is a variant of confined types tailored for the memory management discipline of RTSJ. ScopeJ imposes a naming discipline based on a few annotations and some simple-to-check rules that statically ensure the correct usage of scoped memory, thereby eliminating the need for dynamic checking. Zhao et al. [145] define Implicit Ownership Types for memory management based on ScopeJ's memory model. The key contribution of this approach is that the programmer does not need to specify any type annotations—ownership is implicit, and therefore not a burden to programmers. A complete formal semantics of the approach is also presented. The work of Andreae et al. [9] uses aspects to facilitate a more modular specification of the code dealing with scoped memories. The ScopeJ approach has also been adapted and applied to SCJ (Safety Critical Java Specification) [133].

6.4 Ownership for Security

One of the original motivations for Confined Types was to address security problems found in the Java library, namely, to prevent certain references from escaping their defining scope. More specifically, each instance of Java `Class` has a list of signers that the security architecture uses to determine the access rights of the class at run-time. A leaking reference to this internal data structure caused a security flaw in JDK1.1 that allowed untrusted applets to gain all access rights. The problem boiled down to the fact that an alias to the array containing the signers was leaked, rather than a copy of the array. (For more details, see [137].)

As mentioned in Section 3.7, Confined Types [136] are a syntactically simple approach to achieving a topological restriction similar to what Ownership Types do, but with a package-level granularity instead of object-level granularity. Confined Types were originally designed to prevent security bugs, such as the Java class signer bug, resulting from leaking references to sensitive objects. One further example application is ensuring that references to cryptographic keys do not leak beyond the crypto module [137].

In a variant of confined types adapted to the setting of Enterprise Java Beans, Clarke, Richmond and Noble [47] address the problem of leaking EJB objects without the appropriate wrappers. Without these wrappers, beans could be accessed directly, thereby circumventing the persistency, distribution, and security functionality that would otherwise be in place. The scheme was based on a specification of which classes corresponded to confined and to boundary elements. Checking was performed at deployment time by inspecting the bytecode. Only classes mentioned in the specification file needed to be checked, as in the original Confined Types, and thus classes were protected against unchecked attackers who did not necessarily conform to the discipline.

Naumann and Banerjee [12] use a simplified form of ownership to achieve pointer confinement (a topological constraint) in a class-based language, and use the resulting language to prove certain noninterference results. Similarly, Skalka and Smith [128] present an system using ownership-related notions for secure capability-based programming. Their core protection model is similar to that of Confined Types [86] and Clarke's finitary version of Ownership Types in the object calculus [45].

In one of the few systems that allows owners to vary (without using ownership transfer), Yu, Potter and Xue [99] introduce the *owners-as-downgraders* policy which increases the flexibility of ownership types systems by allowing an object to downgrade or declassify an object's owner, thereby allowing previously protected objects to be accessible beyond what usually would be allowed. In this setting, downgrading can be considered as a special case of intransitive noninterference.

Using a variant of Universe Types, Dietl, Müller and Poetzsch-Heffter [59] present a type system for applet isolation on JavaCard smart card. Their system statically detects firewall violations, which would otherwise be detected only using dynamic checks.

In a quite different setting, Patrignani, Clarke and Sangiorgi [115] apply Ownership Types to the Join calculus [65] in order to enforce certain security properties. They prove that secrets owned by some process cannot be leaked, even against untyped attackers, that is, processes that are not typed using the Ownership Types system (or any other type system).

6.5 Ownership for Object Upgrades

Boyapati et al. [23] describe a quite unexpected application of Ownership Types in the context of persistent object stores. Their approach uses Ownership Types to ensure, in a modular way, that a persistent object store can be efficiently updated without stopping the application. Modularity allows the programmer to locally reason about the correctness of their upgrades. Ownership Types with effects annotations help to provide the desired modularity condition. The system was implemented in the context of the Thor language [93].

6.6 Software Visualisation and Understanding

Ownership has been used in techniques to provide better understanding of system structure and behaviour. Many of these techniques have a visual component, based either on the static or dynamic structure of a system.

The notion of ownership has also been used to introduce structure into the visualisation of systems, both of the evolving object graph [75,74,76] and of a static abstraction of it (Object Ownership Graphs (OOGs)) [3,135].[5] These approaches exploit the hierarchical nature of ownership to enable the structuring of objects in a visualisation along with the collapsing of parts of the object graph that are not the current focus of whoever is performing the visualisation. Ammar and Abi-Antoun [6] perform further work on the use of OOGs in program comprehension. Using a group of programmers unfamiliar with notions of ownership, they were able to show in a statistically significant fashion that the use of OOGs improve programmers' comprehension compared to programmers who just used class diagrams.

Mitchell uses techniques based on ownership and dominators to summarise memory footprints in order to better understand the memory usage of a program [103]. In his approach, each dominator tree captures unique ownership. Trees are connected by specific edges that describe responsibility, *i.e.,* transfer of ownership. A profiling technique aggregates these structures and uses thresholds to identify important aggregates. The notion of ownership graph summarises responsibility, and the notion of backbone equivalence is used to aggregates patterns within trees, generating concise summaries of heap usage. The ultimate goal of this work is to understand where excessive memory usage occurs in large programs, not to produce a type assignment.

Rayside et al. [123] take an alternative approach to finding and fixing memory leaks which is also based on object ownership. Their techniques involves determining the ownership hierarchy of objects, the size of each object, the time interval that each object is allocated, and the time interval that it is active. This information is reported visually to the programmer. In conjunction with five memory management anti-patterns that are identified based on the authors' experience with object ownership profiling, their tool can help the programmer to identify memory leaks. The authors apply their techniques to fix memory leaks in the Alloy IDE (V3).

7 Foundational Calculi

Although most work on Ownership Types is formalised, only a relatively small amount of work has been done on their foundations.

Very early on, Clarke's thesis [39] formalised Ownership Types in terms of Abadi and Cardelli's object calculus [1]. In this setting ownership contexts are separate from objects. Every object has two ownership contexts, namely, a representation context, or storing its representation, and an owner context, which was its owner. Whether an object had permission to access another object was determined using the representation context of the referrer and the owner context of the referee. Clarke gave constraints on the relationships between ownership contexts that guaranteed that the owners-as-dominators property held. The

[5] Interestingly, it was while studying software visualisation that James Noble realised the pressing need for proper alias control, which eventually lead to Flexible Alias Protection.

formalism also supported type parameterisation and existential quantification of owners.

The foundational calculus F_{own} [87] was built on top of system F to provide a foundation for Ownership Domains. To capture the specifics of the system, a special permissions system based on ownership domains and links was added into the calculus. Being based on system F, the calculus also supported parametric polymorphism.

Cameron and Drossopoulou [32,30] study Ownership Types systems that include notions of owner variance and unknown ownership context—we have seen various systems like this above. Their system Jo∃ casts these systems into a uniform setting and removes many of the limitations of the more ad hoc approaches found in the literature. Their calculus supports parameterisation of types and ownership contexts, and allows variant subtyping of ownership contexts using existential types. The explicit use of existential types makes the type-theoretic foundations of owner variance and unknown ownership context more transparent. Building on this work, Cameron, Drossopoulou and Noble's chapter [31] explains Ownership Types in terms of dependent types, elucidating the idea that Ownership Types are actually a kind of phantom type [89], namely, a type that does not contribute to the run-time representation of values.

Using the Fractional Permissions framework [27,28], Zhao and Boyland [149] encode the owners-as-dominators and owners-as-locks models. Fractional Permissions provide a uniform, albeit low-level, view on these two models, and permit the encoding of other models, such as variants of Multiple Ownership, in the same setting.

A similar approach based on encoding into logic is taken by Wang et al. [138] who add a notion of confinement to separation logic. One of the goals of their work is to reason about confinement independently of any particular confinement discipline. In different work, Wang and Qui [139] present a generic model of confinement aimed at breaking the shackles imposed by purely syntactic definitions by providing semantic definitions for encoding various confinement schemes. In some sense their work provides a more complete formal model of ideas proposed by Noble et al. [110].

8 Empirical Studies

A number of empirical studies of Ownership Types and related systems have been performed. These include programming experiments, such as seeing how Ownership Types interact with design patterns or applying Ownership Types to a given code base and determine what changes need to be made; automatic analysis of the run-time object structures of a corpus of programs; and automatic static analysis of a corpus of programs.

The most natural approaches to evaluating Ownership Types are to apply it to an existing code base or to study the interaction between Ownership Types and design patterns.

AliasJava [5], a precursor to Ownership Domains, was evaluated on various library classes and the Aphyds circuit layout application, which consisted of

12.5kLOC. A core 3kLOC was annotated by the Aphyds developer. Most method parameters were annotated with **`lent`** and many return values could be annotated with **`unique`**, indicating that either sharing was absent or temporary. Instances of classes related to circuit elements were, in contrast, shared among many other objects.

In his masters thesis, Hächler applies Universes to an industrial application [70]. The application was the software of a ticket reader for print-at-home tickets, consisting of more than 50kLOC. The process of annotating this system revealed a number of shortcomings of ownership, which required restructuring of the application, and replacing pass-by-reference by pass-by-copy. One interesting proposed extension was the idea of local universes (akin to Clarke and Wrigstad's scoped owners [42]), which allow read and write access within the scope of a pure method in such a way that the mutable effects are encapsulated within the pure method.

An evaluation of ownership domains was done on four real world programs, JHotDraw and HillClimber, each of 15kLOC, Aphyds (8kLOC), and CryptoDB (3kLOC) supported by a reimplementation of the type system and an Eclipse plug-in [3,2]. The case studies identified a few patterns of ownership (e.g., "ownership domains expose tight coupling") and some weaknesses of the existing Ownership Domains type system (e.g., public domains are hard to use, annotations are verbose). More details can be found in Abi-Antoun's PhD Thesis [2]. A further case study considers the amount of effort required to refactor a 16kLOC application (HillClimber) to enforce appropriate architectural constraints expressed with the help of Ownership Domains [3,2]. Again a number of lessons are reported, along with perceived limitations of Ownership Domains. The reader is invited to consult these papers for more details.

In an early exploration of using Ownership Types in practice, Cele and Stureborg [36] found that embedding ownership information in programs made them less flexible, especially for reuse and refactoring. In their qualitative study, balancing flexibility and encapsulation emerged as a key aspect of programming with ownership. A frequent pattern in their programming illustrates this; it relies on subsumption to remove owners from types,[6] such as for call-backs and listeners.

In his master's thesis, Nägeli [109] uses design patterns to evaluate three different Ownership Types systems (Universes, Clarke and Drossopoulou's Joe$_1$, and Ownership Domains). His work identifies numerous difficulties in the various disciplines (and the tool support for them). Based on this study, he lists the following requirements for Ownership Type systems: alias control for representation objects, support for read-only references, multiple ownership, ownership transfer, friend contexts (analogous with friends in C++ [132]). These concepts (apart from friend contexts) have appeared in some form in other systems cited in this survey, but no system includes them all.

The evaluation of OIGJ [150] included a demonstration that their system could express the factory and visitor patterns, and be used to type check the standard `java.util` collections (except for clone methods) without refactoring

[6] It was latter dubbed the "Hide Owner Pattern" [141].

and with only a small number of annotations. Sergey and Clarke's Gradual Ownership Types [127] were empirically evaluated by integrating ownership annotations into a non-generic version of Java's collections framework, resulting in the analysis of about 8,200 lines of code. Only on significant refactoring was required to satisfy the type checker.

The second approach to evaluate Ownership Types is to semi-automatically analyse a large code base—state-of-the-art ownership inference is not good enough to do this fully automatically.

The unified ownership inference framework [79], implemented on top of the Checker Framework,[7] was evaluated experimentally on 110 kLOC, including `ejc`, Eclipse IDE compiler, and `javad`, the Java class file disassembler. The results indicated that that a large amount of non-trivial ownership annotations could be applied in these production-quality applications.

For the Confined Types discipline, Grothoff et al. [69] analysed a large body of code, consisting of over 46,000 classes. Their tool Kacheck/J uncovered 24% confined classes and interfaces. In a language with generics (such as modern versions of Java), this number would increase to 30%. After inferring tighter access control modifiers, this number went up even further to 45%, meaning that 45% of all package scoped classes were confined.

Vanciu and Abi-Antoun's chapter [135] present a significant experimental evaluation of the use of Ownership Domains in practice based on Ownership Object Graphs (OOGs), which incorporate ownership information into an object graph to provide abstractions based on ownership and types. Their approach is based on annotating several systems using Ownership Domains and using static analysis to extract OOGs. They added annotations to 100 KLOC of real object-oriented code. The chapter presents a vast range of statistics, which we won't repeat here. Their conclusion is that ownership can make a significant contribution to expressing designs more abstractly.

The third approach to evaluating the potential of Ownership Types is to study the object-graph structures of running programs. To this end, Potanin, Noble and Biddle [118] employed a tool to take snapshots of the heaps of running programs and applied it to a large corpus of Java programs. The tool computed various metrics on the collected heaps related to notions of uniqueness, ownership and confinement, to determine how often such concepts appear in actual running programs. Their results indicate that such concepts are often used in practice—12% of objects were not uniquely referenced, ownership hierarchies were on average five layers deep, and around one third of objects were referred to only by classes in the same package.

9 Discussion and Conclusion

This chapter has presented a comprehensive survey of many variations, extensions and applications of Ownership Types. It is time now to take a step back

[7] `http://types.cs.washington.edu/checker-framework/`

and consider what remains to be done in the future. What follows is based on criticism of Ownership Types, both published and otherwise, discussions at various forums, and our own experience.

Designing good context-dependent constraints. Ownership Types systems need to be better tailored to specific problems domains. When doing this, a statement of invariants or desired properties must come first, and the type system needs to be designed around those within some framework that is capable of linking types to guarantees. Systems such as Ownership Domains cater for this to some degree, and various Confined Types variants have been specifically tailored for given problem domains.

A lot of work already been done add to address the inflexibility of systems, as witnessed by mechanisms such Ownership Domains' separation of mechanism from policy, External Uniqueness, owner-polymorphic methods, among others. These should be considered as possible ingredients in future Ownership Types systems, but it needs to be clear precisely what properties each new ingredient enforces or violates, in relation to the properties that need to be enforced.

Better integration with dynamic mechanisms. Ownership Type systems should rely on and integrate better with mechanisms that dynamically provide guarantees about run-time behaviour, such as synchronised methods or locks. Just as ownership can be used to reduce the amount of synchronisation, synchronisation strengthens temporary guarantees about exclusive ownership.

Larger scale analysis of ownership usage in existing code bases. More thorough empirical analysis of ownership in real systems needs to be performed, to capture both common usage patterns of ownership and how the use of ownership evolves across time, but also to determine and characterise common alias patterns and problems.

Deployment in practice. Numerous case studies with various systems have been carried out (Section 8), a lot of experience both positive in terms of usage patterns and negative in terms of weaknesses has been gathered, and this experience has fed back into the design of better systems. Nevertheless, experience with Ownership Types would benefit significantly if it were used to build real systems, ideally in a commercial setting. A proper scientific analysis of the benefits should accompany this activity.

Reduced syntactic overhead. Ownership Types often impose a heavy syntactic burden on programmers. Many developments in tool support, appropriate default annotations (though we've said little about this), type inference and other program analysis techniques, have helped reduce this load, but more sophisticated and more integrated approaches are needed. Such approaches may rely heavily on static analysis techniques such as alias- and shape-analysis.

Address the library code problem. A problem common to any specialised type system (not just Ownership Types systems) is that library code is not checked using the type system. This problem can be tackled from two directions. One is to provide tool support and automatic analysis. The other is by using notions such as gradual typing [127], which allow some of the code to be annotated and checked statically, leaving other checks to run-time.

Exploitation of the ideas to better support concurrent programming. Due to the petering out of Moore's law, multicore computers have become mainstream, and increasing amounts of concurrent processing resources are available on the desktop. Programs need to be concurrent to exploit these resources, but reasoning about such programs, for humans and compilers alike, is hard, and annotations describing aliasing and ownership offer relief. More needs to be done to make the ownership annotations fully exploitable by compilers, and sufficiently powerful for programmers when they need them, and unobtrusive when they don't.

Acknowledgements. Thanks to Marwan Abi-Antoun, Jonathan Aldrich, Peter Müller, and the anonymous reviewers for their detailed comments.

References

1. Abadi, M., Cardelli, L.: A theory of objects. Springer (1996)
2. Abi-Antoun, M.: Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure. PhD thesis, Carnegie Mellon University, Available as Technical Report CMU-ISR-10-114 (2010)
3. Abi-Antoun, M., Aldrich, J.: Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In: OOPSLA, pp. 321–340 (2009)
4. Aldrich, J., Chambers, C.: Ownership Domains: Separating Aliasing Policy from Mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
5. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330 (2002)
6. Ammar, N., Abi-Antoun, M.: Empirical Evaluation of Global Hierarchical Object Graphs for Coding Activities. In: Working Conference on Reverse Engineering, WCRE (2012)
7. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark (May 1994), DIKU Rapport 94/19
8. Anderson, Z.R., Gay, D., Naik, M.: Lightweight annotations for controlling sharing in concurrent data structures. In: PLDI, pp. 98–109 (2009)
9. Andreae, C., Coady, Y., Gibbs, C., Noble, J., Vitek, J., Zhao, T.: Scoped types and aspects for real-time Java memory management. Real-Time Systems 37(1), 1–44 (2007)
10. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races. In: OOPSLA, pp. 382–400 (2000)
11. Banerjee, A., Naumann, D.A.: Representation independence, confinement and access control (extended abstract). In: POPL, pp. 166–177 (2002)
12. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a Java-like language. In: CSFW, pp. 253–267 (2002)
13. Banerjee, A., Naumann, D.A.: Ownership transfer and abstraction. Technical report, Computing and Information Sciences, Kansas State University, USA (2003)
14. Banerjee, A., Naumann, D.A.: Ownership: transfer, sharing, and encapsulation. In: Proceedings of the 5th Workshop on Formal Techniques for Java-Like Programs, FTfJP 2003 (2003)

15. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. J. ACM 52(6), 894–960 (2005)
16. Banerjee, A., Naumann, D.A.: State Based Ownership, Reentrance, and Encapsulation. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 387–411. Springer, Heidelberg (2005)
17. Banerjee, A., Naumann, D.A.: State Based Encapsulation for Modular Reasoning about Behavior-Preserving Refactorings. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 319–365. Springer, Heidelberg (2013)
18. Blanchet, B.: Escape analysis: correctness proof, implementation and experimental results. In: Cardelli, L. (ed.) POPL 1998: Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, pp. 25–37 (January 1998)
19. Bocchino Jr., R.L.: Alias Control for Deterministic Parallelism. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 156–195. Springer, Heidelberg (2013)
20. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. Ph.D., MIT (February 2004)
21. Boyapati, C., Lee, R., Rinard, M.C.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA, pp. 211–230 (2002)
22. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, pp. 213–223 (2003)
23. Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H., Richman, S.: Lazy modular upgrades in persistent object stores. In: OOPSLA, pp. 403–417 (2003)
24. Boyapati, C., Rinard, M.C.: A parameterized type system for race-free Java programs. In: OOPSLA, pp. 56–69 (2001)
25. Boyapati, C., Salcianu, A., Beebee, W.S., Rinard, M.C.: Ownership types for safe region-based memory management in real-time Java. In: PLDI, pp. 324–337 (2003)
26. Boyland, J.: Alias burying: Unique variables without destructive reads. Software—Practice and Experience 31(6), 533–553 (2001)
27. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
28. Boyland, J.: Fractional Permissions. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 270–288. Springer, Heidelberg (2013)
29. Buckley, A.: Ownership types restrict aliasing. Master's thesis, Imperial College London, London, UK (June 2000)
30. Cameron, N.: Existential Types for Subtype Variance - Java Wildcards and Ownership Types. PhD thesis, Imperial College London (April 2009)
31. Cameron, N., Drossopoulou, S., Noble, J.: Understanding Ownership Types with Dependent Types. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 84–108. Springer, Heidelberg (2013)
32. Cameron, N., Drossopoulou, S.: Existential Quantification for Variant Ownership. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 128–142. Springer, Heidelberg (2009)
33. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In: OOPSLA, pp. 441–460 (2007)
34. Cameron, N.R., Noble, J., Wrigstad, T.: Tribal ownership. In: OOPSLA, pp. 618–633 (2010)

35. Cardelli, L., Leroy, X.: Abstract types and the dot notation. Technical Report SRC-RR-90-56, Digital Systems Research Center (March 1990)
36. Cele, G., Stureborg, S.: Ownership types in practise. Master's thesis, Department of Computer and Systems Sciences, Stockholm University and Royal Institute of Technology (2005)
37. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA, pp. 519–538 (2005)
38. Chin, B., Markstrum, S., Millstein, T., Palsberg, J.: Inference of User-Defined Type Qualifiers and Qualifier Rules. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 264–278. Springer, Heidelberg (2006)
39. Clarke, D.: Object ownership and containment. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia (2002)
40. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 417–457. Springer, Heidelberg (2011)
41. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: a simple virtual class calculus. In: AOSD, pp. 121–134 (2007)
42. Clarke, D., Wrigstad, T.: External Uniqueness is Unique Enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 176–200. Springer, Heidelberg (2003)
43. 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)
44. Clarke, D.G., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA, pp. 292–310 (2002)
45. Clarke, D.G., Noble, J., Potter, J.M.: Simple Ownership Types for Object Containment. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 53–76. Springer, Heidelberg (2001)
46. Clarke, D.G., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA, pp. 48–64 (1998)
47. Clarke, D.G., Richmond, M., Noble, J.: Saving the world from bad beans: deployment-time confinement checking. In: OOPSLA, pp. 374–387 (2003)
48. Clifton, C., Leavens, G.T., Noble, J.: MAO: Ownership and Effects for More Effective Reasoning About Aspects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 451–475. Springer, Heidelberg (2007)
49. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Sethi, R. (ed.) Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252 (January 1977)
50. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Rosen, B.K. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (January 1979)
51. Cunningham, D., Drossopoulou, S., Eisenbach, S.: Universe Types for Race Safety. In: VAMP 2007, pp. 20–51 (September 2007)
52. Deutsch, A.: Interprocedural alias analysis for pointers: Beyond k-limiting. In: Sarkar, V. (ed.) Proceedings of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation, Orlando, Florida, pp. 230–241 (June 1994)
53. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)

54. Dietl, W., Drossopoulou, S., Müller, P.: Separating ownership topology and encapsulation with generic universe types. ACM Trans. Program. Lang. Syst. 33(6), 20 (2011)
55. Dietl, W., Ernst, M.D., Müller, P.: Tunable Static Inference for Generic Universe Types. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 333–357. Springer, Heidelberg (2011)
56. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
57. Dietl, W., Müller, P.: Runtime universe type inference. In: IWACO 2007: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (2007)
58. Dietl, W., Müller, P.: Object Ownership in Program Verification. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 289–318. Springer, Heidelberg (2013)
59. Dietl, W., Müller, P., Poetzsch-Heffter, A.: A Type System for Checking Applet Isolation in Java Card. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 129–150. Springer, Heidelberg (2005)
60. Dietl, W.M.: Universe Types: Topology, Encapsulation, Genericity, and Tools. Ph.D., Department of Computer Science, ETH Zurich (December 2009); Doctoral Thesis ETH No. 18522
61. Drossopoulou, S., Clarke, D., Noble, J.: Types for Hierarchic Shapes (Summary). In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 1–6. Springer, Heidelberg (2006)
62. Eichberg, M., Kanthak, S., Kloppenburg, S., Mezini, M., Schuh, T.: Incremental confined types analysis. Electr. Notes Theor. Comput. Sci. 164(2), 81–96 (2006)
63. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: PLDI, pp. 13–24 (2002)
64. Flanagan, C., Abadi, M.: Types for Safe Locking. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 91–108. Springer, Heidelberg (1999)
65. Fournet, C., Gonthier, G.: The Join Calculus: A Language for Distributed Mobile Programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
66. Geilmann, K., Poetzsch-Heffter, A.: Modular Checking of Confinement for Object-Oriented Components using Abstract Interpretation. In: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO 2011 (2011)
67. Gordon, D., Noble, J.: Dynamic ownership in a dynamic language. In: Costanza, P., Hirschfeld, R. (eds.) DLS 2007: Proceedings of the 2007 Symposium on Dynamic Languages, Montreal, Quebec, Canada, pp. 41–52 (2007)
68. Greenfieldboyce, D., Foster, J.S.: Type qualifier inference for Java. In: Bacon, D.F., Lopes, C.V., Steele Jr., G.L. (eds.) OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Montreal, Quebec, Canada, pp. 321–336 (2007)
69. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. ACM Trans. Program. Lang. Syst. 29(6) (2007)
70. Hächler, T.: Applying the Universe type system to an industrial application. Master's thesis, ETH Zurich (2005), http://pm.inf.ethz.ch/projects/student_docs/Thomas_Haechler/Thomas_Haechler_MA_paper.pdf
71. Haller, P., Odersky, M.: Capabilities for Uniqueness and Borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010)

72. Harms, D.E., Weide, B.W.: Copying and swapping: influences on the design of reusable software components. IEEE Transactions on Software Engineering 17(5), 424–435 (1991)
73. Herrmann, S.: Confined Roles and Decapsulation in Object Teams — Contradiction or Synergy? In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 443–470. Springer, Heidelberg (2013)
74. Hill, T., Noble, J., Potter, J.: Scalable visualisations with ownership trees. In: TOOLS, vol. (37), pp. 202–213 (2000)
75. Hill, T., Noble, J., Potter, J.: Visualizing the structure of object-oriented systems. In: VL, pp. 191–198 (2000)
76. Hill, T., Noble, J., Potter, J.: Scalable visualizations of object-oriented systems with ownership trees. J. Vis. Lang. Comput. 13(3), 319–339 (2002)
77. Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R.C.: The Geneva convention on the treatment of object aliasing. OOPS Messenger 3(2), 11–16 (1992)
78. Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R.C.: The Geneva Convention on the Treatment of Object Aliasing. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 7–14. Springer, Heidelberg (2013)
79. Huang, W., Dietl, W., Milanova, A., Ernst, M.D.: Inference and Checking of Object Ownership. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 181–206. Springer, Heidelberg (2012)
80. Huang, W., Milanova, A.: Towards efective inference and checking of ownership types. In: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO 2011 (2011)
81. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.K.: Single and loving it: Must-alias analysis for higher-order languages. In: Cardelli, L. (ed.) POPL 1998: Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, pp. 329–341 (January 1998)
82. Bocchino Jr., R.L., Adve, V.S.: Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 306–332. Springer, Heidelberg (2011)
83. Bocchino Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for deterministic parallel Java. In: OOPSLA, pp. 97–116 (2009)
84. Bocchino Jr., R.L., Heumann, S., Honarmand, N., Adve, S.V., Adve, V.S., Welc, A., Shpeisman, T.: Safe nondeterminism in a deterministic-by-default parallel language. In: POPL, pp. 535–548 (2011)
85. Klebermaß, M.: An Isabelle Formalization of the Universe Type System. Master's thesis, Technische Universität München (April 2007)
86. Lindskov Knudsen, J. (ed.): ECOOP 2001. LNCS, vol. 2072. Springer, Heidelberg (2001)
87. Krishnaswami, N.R., Aldrich, J.: Permission-based ownership: encapsulating state in higher-order typed languages. In: PLDI, pp. 96–106 (2005)
88. Kulkarni, A., Liu, Y.D., Smith, S.F.: Task types for pervasive atomicity. In: OOPSLA, pp. 671–690 (2010)
89. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: DSL, pp. 109–122 (1999)
90. Leino, K.R.M., Müller, P.: Object Invariants in Dynamic Contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)

91. Li, P., Cameron, N., Noble, J.: Mojojojo - more ownership for multiple owners. In: International Workshop on Foundations of Object-Oriented Languages, FOOL (2010)
92. Li, P., Nelson, S., Potanin, A.: Ownership for relationships. In: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO 2009, pp. 1–3. ACM, New York (2009)
93. Liskov, B., Castro, M., Shrira, L., Adya, A.: Providing Persistent Objects in Distributed Systems. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 230–257. Springer, Heidelberg (1999)
94. Liu, Y.D., Smith, S.F.: Interaction-based programming with classages. In: OOPSLA, pp. 191–209 (2005)
95. Lu, Y., Potter, J.: A Type System for Reachability and Acyclicity. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 479–503. Springer, Heidelberg (2005)
96. Lu, Y., Potter, J.: On Ownership and Accessibility. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 99–123. Springer, Heidelberg (2006)
97. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL, pp. 359–371 (2006)
98. Lu, Y., Potter, J., Xue, J.: Validity Invariants and Effects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 202–226. Springer, Heidelberg (2007)
99. 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)
100. Ma, K.-K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: Bacon, D.F., Lopes, C.V., Steele Jr., G.L. (eds.) OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Montreal, Quebec, Canada, pp. 423–440 (2007)
101. Milanova, A., Liu, Y.: Practical static ownership inference. Technical report, Rensselaer Polytechnic Institute, Troy NY 12110, USA (2010)
102. Milanova, A., Vitek, J.: Static Dominance Inference. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 211–227. Springer, Heidelberg (2011)
103. Mitchell, N.: The Runtime Structure of Object Ownership. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
104. Moelius III, S.E., Souter, A.L.: An object ownership inference algorithm and its applications. In: MASPLAS 2004: Mid-Atlantic Student Workshop on Programming Languages and Systems (2004)
105. Müller, P., Poetzsch-Heffter, A.: Universes: a type system for controlling representation exposure. In: Programming Languages and Fundamentals of Programming. Fernuniversität Hagen (1999)
106. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming 62, 253–286 (2006)
107. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
108. Müller, P., Rudich, A.: Ownership transfer in universe types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA 2007, pp. 461–478. ACM, New York (2007)
109. Nägeli, S.: Ownership in design patterns. Master's thesis, ETH Zurich (2006), http://pm.inf.ethz.ch/projects/student_docs/Stefan_Naegeli/Stefan_Naegeli_MA_paper.pdf
110. Noble, J., Biddle, R., Tempero, E., Potanin, A., Clarke, D.: Towards a model of encapsulation. In: IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, Darmstadt, Germany (2003)

111. Noble, J., Clarke, D.G., Potter, J.: Object ownership for dynamic alias protection. In: TOOLS, vol. (32), pp. 176–187 (1999)
112. Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
113. Östlund, J., Wrigstad, T.: Multiple Aggregate Entry Points for Ownership Types. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 156–180. Springer, Heidelberg (2012)
114. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, uniqueness, and immutability. In: TOOLS, vol. (46), pp. 178–197 (2008)
115. Patrignani, M., Clarke, D., Sangiorgi, D.: Ownership Types for the Join Calculus. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 289–303. Springer, Heidelberg (2011)
116. Permandla, P., Roberson, M., Boyapati, C.: A type system for preventing data races and deadlocks in the Java virtual machine language. In: LCTES, pp. 1–10 (2007)
117. Poetzsch-Heffter, A., Geilmann, K., Schäfer, J.: Infering Ownership Types for Encapsulated Object-Oriented Program Components. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 120–144. Springer, Heidelberg (2007)
118. Potanin, A., Noble, J., Biddle, R.: Checking ownership and confinement. Concurrency and Computation: Practice and Experience 16(7), 671–687 (2004)
119. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Featherweight generic confinement. J. Funct. Program. 16(6), 793–811 (2006)
120. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for Generic Java. In: OOPSLA, pp. 311–324 (2006)
121. Potanin, A., Östlund, J., Zibin, Y., Ernst, M.D.: Immutability. In: Clarke, D. (ed.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 233–269. Springer, Heidelberg (2013)
122. Potter, J., Noble, J., Clarke, D.G.: The ins and outs of objects. In: Australian Software Engineering Conference, pp. 80–89 (1998)
123. Rayside, D., Mendel, L.: Object ownership profiling: a technique for finding and fixing memory leaks. In: ASE, pp. 194–203 (2007)
124. Schaefer, J., Poetzsch-Heffter, A.: A parameterized type system for simple loose ownership domains. Journal of Object Technology 6(5), 71–100 (2007)
125. 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)
126. 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)
127. Sergey, I., Clarke, D.: Gradual Ownership Types. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 579–599. Springer, Heidelberg (2012)
128. Skalka, C., Smith, S.F.: Static use-based object confinement. Int. J. Inf. Sec. 4(1-2), 87–104 (2005)
129. Spring, J.H., Pizlo, F., Privat, J., Guerraoui, R., Vitek, J.: Reflexes: Abstractions for integrating highly responsive tasks into Java applications. ACM Trans. Embedded Comput. Syst. 10(1) (2010)
130. Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: Streamflex: High-throughput stream programming in Java. In: OOPSLA, pp. 211–228 (2007)
131. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)

132. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
133. Tang, D., Plsek, A., Vitek, J.: Static checking of safety critical Java annotations. In: JTRES, pp. 148–154 (2010)
134. Tofte, M., Talpin, J.-P.: Region-based memory management. Inf. Comput. 132(2), 109–176 (1997)
135. Vanciu, R., Abi-Antoun, M.: Object Graphs with Ownership Domains: An Empirical Study. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. LNCS, vol. 7850, pp. 109–155. Springer, Heidelberg (2013)
136. Vitek, J., Bokowski, B.: Confined types. In: OOPSLA, pp. 82–96 (1999)
137. Vitek, J., Bokowski, B.: Confined types in Java. Softw., Pract. Exper. 31(6), 507–532 (2001)
138. Wang, S., Barbosa, L.S., Oliveira, J.N.: A relational model for confined separation logic. In: TASE, pp. 263–270 (2008)
139. Wang, S., Qiu, Z.: A generic model for confinement and its application. In: TASE, pp. 57–64 (2008)
140. Wren, A.: Inferring ownership. Master's thesis, Imperial College London, London, UK (June 2003)
141. Wrigstad, T.: Ownership-Based Alias Management. PhD thesis, Royal Institute of Technology, Kista, Stockholm (May 2006)
142. Wrigstad, T., Clarke, D.: Existential owners for ownership types. Journal of Object Technology 6(4), 141–159 (2007)
143. 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)
144. Smith, S., Liu, Y.D.: Pedigree types. In: IWACO 2008: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (2008)
145. Zhao, T., Baker, J., Hunt, J., Noble, J., Vitek, J.: Implicit ownership types for memory management. Sci. Comput. Program. 71(3), 213–241 (2008)
146. Zhao, T., Noble, J., Vitek, J.: Scoped types for real-time Java. In: RTSS, pp. 241–251 (2004)
147. Zhao, T., Palsberg, J., Vitek, J.: Lightweight confinement for featherweight Java. In: OOPSLA, pp. 135–148 (2003)
148. Zhao, T., Palsberg, J., Vitek, J.: Type-based confinement. J. Funct. Program. 16(1), 83–128 (2006)
149. Zhao, Y., Boyland, J.: A fundamental permission interpretation for ownership types. In: TASE, pp. 65–72 (2008)
150. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in Generic Java. In: OOPSLA, pp. 598–617 (2010)