Ownership Types for Local Reasoning

CS5218: AY2014/2015 Semester 2, Final Project

Darius Foo
A0097282@u.nus.edu

Daryl Seah
A0026468@u.nus.edu

Joel Low
A0097630@u.nus.edu

ABSTRACT

The use of ownership types improves local reasoning, allowing both programmers and program analysis tools to reason more comprehensively about the correctness and behaviour of programs. This would improve the scalability of program analysis tools. This paper examines four variations of ownership types, and compares their expressiveness as well as their ability to achieve the goal of improving local reasoning. However, owing to the difficulty of annotating programs, it is likely that ownership types will only be used in safety-critical programs.

1. INTRODUCTION

The advent of Object-Oriented Programming has allowed programmers to write and maintain larger components and programs, owing to the modularisation brought about by encapsulation. This increased modularisation of programs ought to result in a corresponding ease of comprehension for programmers, since programmers are now able to reason about programs at a modular level. Likewise, it should follow that static program analysis tools be able to improve the speed at which analyses can be completed, especially when analysing a large program.

In practice, many escape mechanisms which break the encapsulation of an object are used. These mechanisms, while allowing expressiveness in programs, prevent the programmer from reaping the benefits of being able to isolate a module and reason about its behaviour and effects. We present an example in listing 1, originally described in [16].

Here we present a car with a driver and an engine. The car defines a `start` method which will start the engine, but only if a driver is present. However, due to the lack of encapsulation, the car's engine can be started directly in line 22, bypassing the check. This should not be allowed because it breaks encapsulation.

The traditional means of preventing this is to utilise access specifiers: using Java's syntax, there would be four levels of access: `public`, `protected`, `private`, and default (package-level) access. These access specifiers are annotations on types and variables, limiting *visibility* of the identifiers of said types and variables to the appropriate scope. Listing 2, modified from listing 1, demonstrates their use, but also that enforcing encapsulation using access specifiers alone is insufficient.

In listing 2 we annotate each method and member variable with an appropriate access specifier. We also define an additional *accessor* method, for the situation where a property of the car's engine must be accessed outside of the car (such as an instrument attached to the car's electronics to measure performance.) However, as seen on line 26, the original problem has not been resolved: the engine can still be started externally. There is no way to reconcile access

```
1  class Person {}
2
3  class Engine {
4    void start() { /* ... */}
5  }
6
7  class Car {
8    Engine engine;
9    Person driver;
10
11   void start() {
12     if (driver != null) {
13       engine.start();
14     }
15   }
16 }
17
18 class Main {
19   static void main() {
20     Car car = new Car();
21     car.start();
22     car.engine.start();
23   }
24 }
```

Listing 1: Car

to parts of an object based on who 'owns' the object, and who is accessing the object. Thus, access specifiers alone are not expressive enough.

While access specifiers can be modified to support such a scenario[1], the resulting API is clumsy because every time a type is defined, programmers must manually define the behaviours allowed by mutable and immutable references to the object. There would consequently be an explosion of types, making reasoning about the program more difficult.

In this report, we describe and evaluate how ownership types can be leveraged for local reasoning in the analysis and verification of programs. Section 2 provides an introduction to concept of ownership types, the core principles and terminology used, and its various applications in program analysis. Section 3 gives a detailed overview of each of the four ownership type systems that we have investigated in depth. Section 4 provides a comparative evaluation on the expressiveness of each system and their benefits for local reasoning and analysis. Finally, we conclude in section 5 with a summary of the fundamental limitations of ownership types. Our coverage of the topic in breadth is spread over sections 2 and 3.

[1]Read-only and mutable interfaces can be extracted from the class, and the appropriate interface returned from methods. This is not common practice; however, the Cocoa runtime for Objective-C is notable for doing this.

```
1  class Person {}
2
3  class Engine {
4    public void start() { /* ... */ }
5  }
6
7  class Car {
8    private Engine engine;
9    private Person driver;
10
11   public void start() {
12     if (driver != null) {
13       engine.start();
14     }
15   }
16
17   public Engine getEngine() {
18     return engine;
19   }
20 }
21
22 class Main {
23   public static void main() {
24     Car car = new Car();
25     car.start();
26     car.getEngine().start();
27   }
28 }
```

Listing 2: Car with Access Specifiers

```
1  class Person {}
2
3  class Engine<EngineOwner> {
4    void start() { /* ... */ }
5  }
6
7  class Car<CarOwner, DriverOwner> {
8    Engine<this> engine;
9    Person<DriverOwner> driver;
10
11   void start() {
12     if (driver != null) {
13       getEngine().start();
14     }
15   }
16
17   Engine<this> getEngine() {
18     return engine;
19   }
20 }
21
22 class Main {
23   static void main() {
24     Car<this, world> car = new Car<this, world>();
25     car.start();
26     car.getEngine().start();
27   }
28 }
```

Listing 3: Car with Ownership Types

2. OWNERSHIP TYPES

Ownership Types enhance the type system by allowing ownership information to be stored together with type information; this directly overcomes the limitations of only using access specifiers. Using SafeJava's [4] syntax, the Car example can be expressed as per listing 3 above.

Listing 3 introduces the notion of an ownership *type*. Just like generics, this is information which is created and enforced statically. All types are now parameterised by an ownership type, preventing objects with different owners from being assigned (and indeed, referenced). Furthermore, it is not possible to express the ownership type of an object belonging to another object, preventing the encapsulation violation on line 26.

Ownership parameters can also be used within the class declaration, as shown on line 7. Special keywords have been introduced: `world` indicating that the value is not owned by any object (i.e. it is owned by the system), and `this` has been overloaded to refer to the current object's ownership context.

Type-checking this program results in a type-check failure because it is not possible to express the ownership context of the car outside of the class. However, it is possible to express the car's ownership of objects used by it through the ownership parameter mechanism.

This is one of the few approaches that we have examined. The other approaches are conceptually similar; the difference among them lies mainly in their expressivity.

2.1 Definitions

In this paper, we try to make our definitions as generic as possible, allowing a consistent set of terms to be used across the different systems proposed.

Expressivity How complex a notion of ownership a system can allow a programmer to express. We have designed various use cases which encompass a variety of design patterns found in programs used today, and analysed each system for its ability to express such a design pattern. Our use cases can be broadly categorised as such:

- **Abstract Data Types**: We modified a Queue example from [4] and added iterators, which can be seen as a view over the data stored in the data type.
- **Inheritance**: A recurring pattern in Object-Oriented Programming where new classes are augmented with behaviour from existing classes.
- **Generics**: Building upon the ADT use case, Generics allow programmers to take advantage of parametric polymorphism to write reusable components.
- **Cardinality of Ownership**: Having multiple owners for an object is useful for writing parallel/distributed algorithms.
- **Ownership Transfer**: Allowing the transfer of ownership from one object to another is useful in the producer-consumer pattern.

Representation Invariant A constraint on the state of an object, maintained throughout its lifetime.

Owner An object or set of objects responsible for maintaining the representation invariant of an object.

Context *a.k.a. Universe, Box* A nested partition of the object store containing all the objects constituting the representation of a particular object.

2.2 Ownership Topologies

An ownership topology describes the structure and organisation of the object store. In this paper, two topologies are examined:

Hierarchical *a.k.a. Tree* This is a single ownership topology, where every object as one owner.

Directed Acyclic Graph (DAG) This is a multiple ownership topology, where an object can have multiple owners. All owners of the object can modify its invariant.

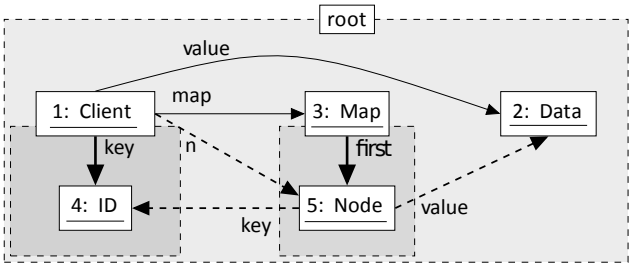


Figure 1: Owners-as-Dominators vs. Owners-as-Modifiers

2.3 Encapsulation Disciplines

Where an ownership topology *describes* the structure of the object store, an encapsulation discipline *prescribes* the structure of the object store. They can thus be seen as related ideas. In this paper, two main encapsulation disciplines are discussed:

Owners-as-Dominators Objects can only be modified by following a reference chain from the root context through the object's owner. Aliasing is restricted, because no aliases to an object B owned by an object A can exist outside of the ownership context of object A.

Owners-as-Modifiers Objects can only be modified by following a reference chain from the root context through the object's owner. However, read-only access is granted in all other contexts. Aliasing is *not* restricted.

Figure 1 (taken from [18]) illustrates the difference between both encapsulation disciplines. References between objects are represented by arrows. The owner of an object is the class at the top of the dashed box the object is in.

If the **owners-as-dominators** discipline is adhered to, then only the solid arrows are allowed. The dashed arrows are references which do not pass through the object's owner, and as such are only allowed in the **owners-as-modifiers** discipline. In the **owners-as-modifiers** discipline, while such a reference is legal, these references cannot be used to mutate the state of the object being referenced.

2.4 Applications

2.4.1 Object Encapsulation

Deriving directly from the example problem given in our introduction, ownership types can be used to improve the expressibility of the concept of ownership [16]. This allows classes to restrict modification of their representation invariants by external code.

Therefore, when programmers are given a new code base annotated with ownership types, they can be ensured that no external code is able to modify the classes' invariants when they are understanding the code. This would allow them to locally reason about the function and behaviour of the module.

The same benefit would also apply for static analysis tools. Because changes can be effected only from one part of the program, a tool need only reason locally to form sound conclusions. This would allow such tools to run in parallel on different parts of the program, allowing even complex analyses to scale with the size of the code base.

2.4.2 Alias/Access Control

If programs adhere to the **owners-as-dominators** encapsulation discipline, then it is not possible to reference objects within another object's representation. Therefore, aliases would be restricted to the scopes where references are allowed [4, 8, 16].

If programs adhere to the **owners-as-modifiers** encapsulation discipline, then there is no alias control; however, it is still not possible to modify the representation of an object if the current object is not its owner [17, 18].

In both cases, reasoning about side-effects are easy. In the former case, it is possible to increase the precision of alias analyses because aliasing is restricted to scopes where the reference is valid.

2.4.3 Data Race/Deadlock Prevention

Because ownership types impose a structure on the object store, it is possible to prevent data races. Typically, when an object **A** needs to be locked for an update, it is not obvious which dependent objects (references which **A** maintains) need to be locked as well. Since there is an ownership topology imposed on the object store, it would be possible to know at compile-time which objects need to be locked as part of locking **A**. This prevents data from being accessed if an object is left unlocked by human error; data races can thus be prevented.

In the same vein, it is possible to prevent deadlocks from occurring by always acquiring a set of locks in the same order. Acquiring multiple locks in the same order prevents a circular wait condition, which is a necessary condition for a deadlock to occur [32]. Regardless of topology, locking objects using depth-first or breadth-first traversal guarantees that all objects which are reachable will be locked. Furthermore, because the traversal is deterministic, it is guaranteed that the order of lock acquisition is always the same [5].

2.4.4 Persisted Object Upgrades

Very often programs need to persist state to disk. The structure of this state might vary between versions of the same program. A problem is presented to the programmer: how to use the data (in the old format) in the updated program? Typically programmers will write transform functions which accept the old program state as input, and produces the corresponding new program state. This works well for simple objects; however, when the object graph is large, or contains cycles, the order to execute these transform functions may not be obvious.

Ownership types enforce constraints on the structure of the object graph. This allows the programmer to write isolated transform functions that transforms only one type in the object graph. When used together with a transactional memory store, it is possible to individually transform objects in any order, so long as all objects in the graph are eventually transformed. When the transaction is committed, it may be statically guaranteed that the new object graph would be valid [4, 6].

3. OWNERSHIP TYPE SYSTEMS

In this section we give an overview of the four ownership type systems that we have investigated in-depth. The papers covered are by no means the entirety of the work in the area. There are numerous other papers on extensions, improvements and related developments [15]. However, we chose these systems because they contributed important ideas and are representative of the gamut of work in ownership types. Each system will be presented individually with a full cross-comparison provided in section 4.

3.1 Clarke's Ownership Types

The concept of ownership was first introduced by Clarke et al. [13, 16] with the intention of mitigating the problems of uncontrolled aliasing in Object-Oriented languages. This was a refinement of a similar idea proposed by Clarke's thesis advisors called *flexible alias protection* [30] which, in turn, took inspiration from Islands [26] and Balloons [3].

Islands and Balloons were early attempts at solving the same problem but did not provide sufficient expressivity and ease-of-use to be practical. For example, an object cannot be stored in two fully-encapsulated containers simultaneously since a stored object would be considered a part of the container's island/balloon and cannot be externally aliased. To mitigate this problem, dynamic aliases (i.e. temporary stack-based references) into a representation are permitted in both these works. Such aliases are read-only in Islands, but have no restrictions in Balloons. Thus, while this escape mechanism solves the problem, it can also expose the representation of the container itself (e.g. nodes in a link-list) and provide no encapsulation in the case of Balloons. Moreover, the authors of [30] argue that it would be hard for programmers to use Balloons because of the complex abstract interpretation required.

Ownership types solve the problem by providing a more flexible/intuitive syntax to the programmer for defining a hierarchical topology over objects without sacrificing encapsulation. The idea is to annotate types with modifiers that define which context (i.e. owner) each reference belongs to, which include:

- `norep`: Not part of the representation of any object.
- `rep`: Owned by `this` object and is part of its representation.
- `owner`: Owned by the owner of `this` object.
- *v*: A context parameter.

Context parameters are declared on classes with a syntax similar to Java's generics, e.g. `class List<elementOwner> { ... }`, and are similarly applied on types, e.g. `rep List<norep> list;`. They allow owners higher in the hierarchy to declare their ownership of references held by descendants in their contexts. In the case of the previous `list` example, the `list` is declared to be part of the client object's context, but the elements it contains has no owner. Parameterised ownership is the key innovation that (partially) solves the container problem and also enables different roles for different references of the same object class.

These declarations and parameters describe a hierarchical structure of object representations that can be easily verified at compile-time. If well-typed, the programmer can be assured that there are no external aliases to any of the objects within a particular context, guaranteeing encapsulation. In particular, Clarke's ownership types enforces the **owners-as-dominators** discipline, which ensures that all paths to an object must pass through its owner.

However, ownership types as originally presented in [16] still had limitations. For example, rules for inheritance/subtyping were yet to be developed and the encapsulation discipline was still too strict to be of practical use (e.g. iterators were not possible.) Addressing these and other limitations was the main focus of subsequent work on the subject, a few of which are covered in the following sections [4, 6, 8, 17, 21].

3.2 Boyapati's SafeJava

Boyapati et al. tackled the expressivity problem with the intention of making ownership types a practical solution for local reasoning and software upgrades in persistent object stores [6]. His idea was to simply provide an exception for inner-classes (which iterators are often implemented with) to directly access and manipulate the representation of their outer-class regardless of ownership, while retaining the **owners-as-dominators** discipline in all other cases.[2]

The declarations and keywords used are also slightly different, but are no less expressive than Clarke's. The first context parameter of a type is taken to be primary owner, eliminating the need for separate `norep`, `rep` or `owner` modifiers and making the syntax more consistent. Context arguments can be one of the following:

- `this`: Referencing `this` object as the owner.
- `world`: Referencing the root context as the owner.
- *v*: A parameterised owner.

For example, the `list` in section 3.1 would be declared as `class List<listOwner, elementOwner> { ... }` and `List<this, world> list;` in Boyapati's syntax.

Boyapati eventually developed his version of ownership types into a system called SafeJava [4] with additional features supporting not just software upgrades and local reasoning but also data race/deadlock prevention and region-based memory. These include:

- **Inheritance** and subtyping rules for ownership.
- **Constraints** on classes and methods with `where` clauses, e.g. `class Container<o, a, b, c, d> where b <= c { /* b must be identical to or an ancestor of c */ }`.
- **Effects** clauses with the `reads` and `writes` keywords declaring which variables are read and written by a method.
- **Transfer** of ownership from one context to another, enabled by declaring such references to be `unique`.

While impressive overall, we find that the "principled violation of encapsulation" as the authors call it) for inner-classes is not a particularly elegant solution.[3] Moreover, binary operations such as `list1.addAll(list2)` and other patterns are also still problematic to express in both Clarke's and Boyapati's systems.

3.3 Dietl and Müller's Universe Types

Universe Types [17] is an ownership system that takes a somewhat different approach to the problem. Influenced by Clarke's original work on ownership [16] and Müller's concept of Universes [28], Dietl et al. developed Universe Types for the purpose of local reasoning in program verification [17, 18, 21].

Their key insight is that restricting aliasing is not really necessary for program verification — it is sufficient to simply restrict the references that may modify an object to those held by objects within the owner's context [21], and permit read-only access in all other cases. In other words, to enforce the **owners-as-modifiers** discipline instead of owners-as-dominators, thereby giving programmers greater expressibility. For example, read-only iterators and binary operations are not an issue in Universe Types.[4]

Instead of the fully context parametric approach of Boyapati [4], which Clarke also adopted [11], Dietl et al. proposed using type modifiers exclusively for the declaration of ownership. There are three programmer-expressible modifiers in Universe Types:

[2] Clarke actually proposed a similar idea in his thesis [11] but with a discipline that is not as useful for local reasoning [6].
[3] In fact, Aldrich and Chambers have generalized such an approach in their development of *ownership domains* [1]. We did not have time to evaluate this system in depth, but it appears to be *too* expressive for programmers to use easily in practice.
[4] Mutable iterators are still a problem.

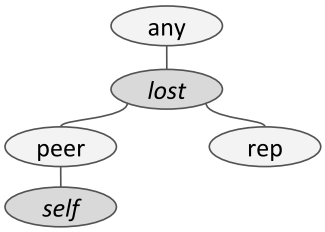


Figure 2: Ordering of Modifiers in Universe Types.

- **any**: No restrictions on the owner. Similar to Clarke's norep.
- **rep**: Owned by this object and is part of its representation.
- **peer**: Owned by the owner of this object. Similar to Clarke's owner.

In addition, there are two non-expressible modifiers that are required for type-checking:

- **lost**: There are constraints on the owner that cannot be expressed, i.e. the identity of the owner has been lost.
- **self**: Refers to this object, which is a specialisation of peer.

The modifier lost can be the result of following a reference chain and attempting to adapt the declared modifiers of the references to the viewpoint of the current object. For example, following a 2-reference chain declared any.rep would result in lost because rep requires a specific owner to type-check but the ownership of its parent object is not available.

These five modifiers form a hierarchy as shown in fig. 2. An object assignment (e.g. $\alpha = \beta$) is legal if the ownership type of α is expressible and is identical to or a generalization of the type of β. Note that nothing can be assigned to a lost reference since the ownership information required for verification is missing, but we can assign anything to any (including lost) since the ownership information will ultimately be discarded.

An effects system in the form of **pure** and **impure** annotations on methods is also supported. Methods are declared pure if they have no side-effects (i.e. do not change the state of the heap) and impure otherwise. This is necessary since external aliases to an encapsulated object are read-only and therefore only pure methods may be invoked on such references.

Universe Types is clearly limited without some form of ownership parameterisation similar to Clarke's [11] and Boyapati's [4]. However, when Clarke and Boyapati developed their ownership systems, Java had yet to introduce generic classes. Dietl et al. subsequently developed Generic Universe Types [18,20,21], which extended Universe Types to languages that have a generic type syntax similar to Java's. As it turns out, unlike [11] and [4], which conflict with the syntax of Java's generic classes, the approach taken with Universe Types (in applying modifiers to existing types) extends naturally to the generic type syntax of Java and adds the ownership parameterisation necessary to make the system practical. An example of the ownership declarations for an imaginary set of generic collection classes (adapted from [20]) is provided in listing 4.

Generic Universe Types does have limitations, but it seems to be one of the more popular and practical ownership systems in existence, likely due to its natural compatibility with the current syntax of Java and (relative) simplicity. Universe Types has been extended with ownership transfers [29], and both Universe Types and Generic Universe types feature type inference [22]. At least two implementations of a type checker [10, 19] and one for inference [19] are available as well.

```
1  class Link<X> {
2    X next;
3  }
4
5  class Node<K,V> extends Link<peer Node<K,V>> {
6    K key; V value;
7  }
8
9  class Iterator<X extends any Link<X>> {
10   X current;
11
12   Iterator(X first) {
13     current = first;
14   }
15
16   pure boolean hasNext() {
17     return current != null;
18   }
19
20   impure X next() {
21     X result = current;
22     current = current.next;
23     return result;
24   }
25 }
26
27 class Map<K,V> {
28   rep Node<K,V> first;
29
30   impure peer Iterator<rep Node<K,V>> iterator() {
31     return new peer Iterator<rep Node<K,V>>(first);
32   }
33 }
```

Listing 4: Usage of Generic Universe Types

3.4 Cameron's Multiple Ownership

The previous type systems impose rigid object topologies well, but suffer in expressivity: the hierarchical model of ownership cannot accommodate certain object structures. For example, listing 5 (visualised in fig. 3), which is originally described in [8], is impossible to express.

The object topology presented here is a DAG instead of a tree: both Projects and Workers may conceptually own Tasks and accordingly modify them. In other words, the axis of organisation is flexible and may be either organised by Projects or by Workers.

The **owners-as-dominators** discipline prevents this, as it requires that the object topology be a tree: only one owner may dominate access paths to Tasks. The **owners-as-modifiers** discipline also causes problems as previously described: either Projects or Workers may own tasks, so only one would be able to modify them.

To work around this problem with single ownership, the object structure would have to be flattened, making all the objects involved peers of each other. This would solve the immediate problem, but would remove all encapsulation and defeats the purpose of having an ownership type system in the first place.

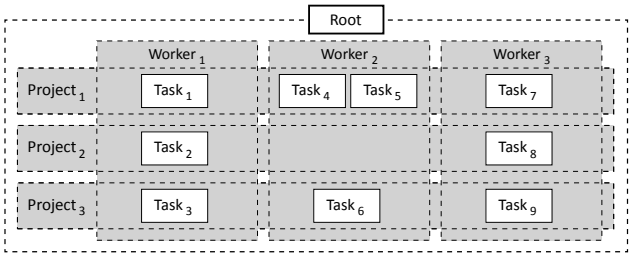


Figure 3: Multiple Ownership

```
class Task { /* ... */ }
class TaskList { /* ... */ }

class Project {
  TaskList tasks;
  void delay() {
    for (Task t : tasks) { t.delay(); }
  }
}

class Worker { // Symmetric
  TaskList tasks;
  void delay() {
    for (Task t : tasks) { t.delay(); }
  }
}
```

Listing 5: Multiple Ownership

```
class Project {
  void add(Task<this & ?> t) { /* ... */ }
}
new Project().add(new Task<p1 & w1>());
```

Listing 6: Existential Owners

These deficiencies motivated [8], which generalises the notion of single ownership by reimagining ownership contexts (called boxes in their terminology) as sets; multiple or joint ownership is then easily expressible via set intersection.

The **owners-as-dominators** discipline is too constraining as there are multiple access paths to objects – it is therefore relaxed. The **owners-as-modifiers** discipline still applies in principle, allowing any one of an object's multiple owners to modify it.[5] Multiple ownership is reified with several syntactic additions:

- **Ownership parameters** (as in [4]), are used to express secondary or child owners.
- **An intersection operator** (`&`) on ownership contexts is introduced. It specifies that an object's owner is the intersection of two existing ownership contexts: conceptually, if an object's owner is `a & b`, it means that the object is jointly owned by a and b.
- **Wildcards** (or *existential owners*) allow programmers to express the notion of co-owners which the current scope does not have access to. This is illustrated in listing 6.[6]

The ability to specify intersection and disjointness constraints is supported by the **`intersects`** and **`disjoint`** keywords.

- `a intersects b` is required for `a & b` to type-check. This forces the programmer to specify upfront which ownership contexts intersect, reducing the chance of erroneous or invalid ownership parameterisation.
- `a disjoint b` guarantees effect-independence between a and b. This can be seen intuitively as we are sure that the ownership contexts of a and b do not intersect, and thus method calls on a and b cannot interfere.

[5]In practice, however, the implementation of multiple ownership discussed in the paper, MOJO, does not enforce owners-as-modifiers; it has been left to future work.

[6]Ownership parameters as specified in the paper *are* covariant; a notable difference from Java's generic type parameters, which are *invariant*. `Task<a & b>` is considered a subtype of both `Task<a & ?>` and `Task<b & ?>`, allowing it to be used wherever the latter two are expected.

```
class C<owner, a, b> where
  a intersects owner,
  b disjoint owner
{
  // legal: known intersection
  Object<a & owner> o1;

  // illegal: known disjointness
  Object<b & owner> o2;

  // illegal: nothing known
  Object<a & b> o3;
}

class Example {
  void method(Object a, Object b, Object c) {
    // legal
    new C<a, b, c>();
    // illegal: disjointness is irreflexive
    new C<a, b, a>();
  }
}
```

Listing 7: Multiple Ownership Constraints

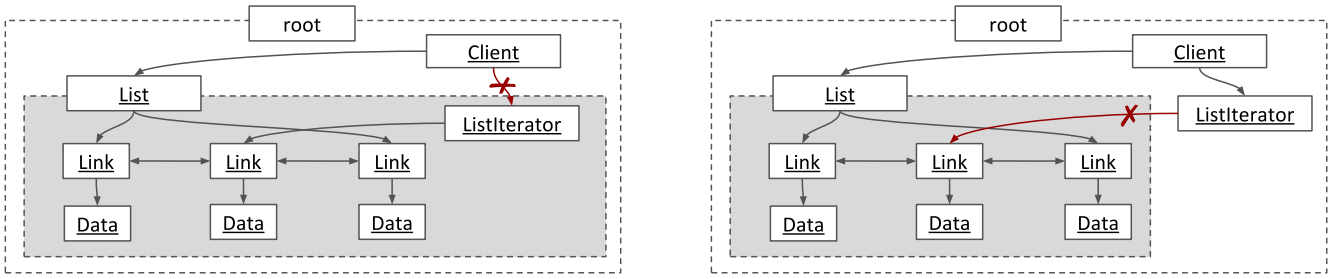
- **`where`** clauses enable us to constrain ownership parameters to classes. This is illustrated in listing 7: we are unable to take the intersections of disjoint ownership boxes, or boxes that we know nothing about. Another notable constraint is that disjoint boxes cannot be parameterised with identical elements as disjointness is irreflexive.

Multiple ownership as described is a powerful notion but has a number of limitations:

- **The owners-as-dominators discipline is dropped**. Access paths are no longer restricted to one dominating object, and thus the type system cannot make guarantees about object encapsulation that are nearly as strong. Cameron et al. propose future work [8] on extending **owners-as-dominators** to accommodate multiple owners, ensuring that at least *one of the owners* of an object dominates access paths to it.
- **Ease of use may suffer**. Ownership constraints reflect their implementation, which in this case is in the form of sets. Their semantics are difficult to reason about without a working understanding of the view of ownership contexts as sets, and may not be as easy to use as the other type systems.
- **Multiple ownership has not been fully generalised** to solve other problems that ownership types are known for, or already have solutions for. More specifically, work on integrating it with generics and race detection is ongoing. This is understandable as it is work in a new direction, being the first implementation to relax the **owners-as-dominators** discipline and admit multiple owners.

4. EVALUATION

In this section we evaluate each of the systems that we were introduced in section 3. We first examine the expressivity of each system, because a system that is too restrictive would prevent widespread adoption. We then look at the unique features that each system presents, and what that would allow in programs. Finally, we examine the benefits that adopting such a system would bring.



(a) Inside Representation

(b) Outside Representation

Figure 4: List Iterator Impossible to Define with Owners-as-Dominators

4.1 Expressivity

As mentioned earlier, we have chosen to measure expressivity in terms of the design patterns and object structures that the different systems support.

Full code examples can be found in the `examples/comparisons` directory. Each code example includes a standard Java implementation (called `standard.java`), and an equivalent implementation for each of the systems we have examined. Systems which did not allow expressing the example would have a comment indicating so. If a system does not allow expressing a given syntactic construct (e.g. inheritance or inner classes), then the implementation for that system would be omitted.

4.1.1 Collection Views

A common design pattern when designing abstract data types is to define *views*. A view is an object that presents the data in some other object – iterators over a container being the most common view used in programs today. These views may allow modification of the object that it is associated with. Owing to its ubiquity, we have adopted it as a benchmark to assess the expressivity of ownership types.

Views generally require access to the container's representation invariant to be efficient. As a case in point, assume that an iterator over a linked list does not have access to its links and nodes. While the iterator would still be able to traverse the linked list (using the public interface that the collection presents; for example, accessing elements by index), accessing successive elements in the list could require $O(n)$ time, resulting in a complete list traversal that requires $O(n^2)$ time.

Even more problems are presented if the views need to be mutable. For example, Java's iterators allow removal of the next element during traversal [31].

Owing to the variety of possible views, we have written three different Iterator implementations:

1. **Immutable views** (`immutable-view`): This example involves a linked list and an iterator which is not allowed to modify the list that it is associated with.
2. **Mutable views** (`mutable-view`): This example involves a linked list and an iterator which can `remove()` the element that the iterator points to. This mirrors Java's implementation of iterators.
3. **Mutable views as Inner classes** (`mutable-view-inner-class`): This example is the same as the Mutable views example, except that the iterator is implemented as an inner class of the linked list.

This category of simple examples is able to articulate the differences between the various systems very well. The **owners-as-dominators** discipline, following the definition given in section 2.1 was shown to be too strict to be able to capture the notion that an object is able to explore the representation invariant of another, even in a controlled way. Figure 4 highlights the limitations well: following the discipline, it would not be possible to define an iterator that can be used both by client code (represented by the `Client` class) and have simultaneous access to the `List`'s representation invariant.

The **owners-as-modifiers** discipline allowed read-only references to other objects' representation invariants. This was sufficiently liberal to implement an iterator over the container. However, this was not particularly useful when the iterator needed to mutate the container, as in the Mutable views example.

Therefore, the Mutable views as Inner classes example was constructed to capitalise on special privileged access [4] or multiple ownership [8]. It turns out that both systems did allow an efficient view to be written.

4.1.2 Binary Operators

Another common pattern that occurs frequently in programs is a binary operator on values, such as the *append* operation on two lists. The two values may have different owners, such as when merging the results of a parallel reduction function. We have chosen to implement a queue which allows merging together to illustrate this situation.

Typically, in an object-oriented program, a binary operator is implemented as a method of the values' class and invoked with one value as the receiver. The actual operation is performed with respect to the recipient. It follows that the recipient must have access to the internals of the other in order to carry out the operation. If there are any changes in the state of the objects, they will typically be to the recipient, hence read-only access to the other is sufficient. We will also assume that this access is not granted via an access specifier in order to retain encapsulation.

Once again, the **owners-as-dominators** discipline is proving to be too strict to allow such an operation. Figure 5 demonstrates that it is not possible for either object to access each other's representation invariants. The **owners-as-modifiers** discipline will not prevent a programmer from implementing such an operation as read access is sufficient.

Universe Types allows objects with a `peer` [21] relationship to explore each other's representations. Therefore, binary operators can be implemented even if `peer` access is not granted.

Multiple ownership also allows binary operators to be implemented, with the **owners-as-dominators** discipline relaxed. This

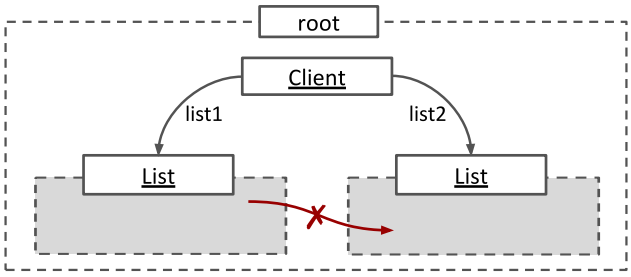


Figure 5: Binary operators do not have access to another object's representation invariant

can be accomplished by declaring the operator as accepting objects that the type has co-ownership for. Read access is granted even with the **owners-as-modifiers** discipline, as the operands to the operator appear in the same ownership tree.

4.1.3 *Ownership Transfer*

A common design pattern in concurrent programming is the producer-consumer pattern. In this pattern, two tasks run in parallel, with the output of the first being used as input in the second. This would exploit increased parallelism in modern computers. If ownership cannot be transferred from producers to consumers, then possible operations would be restricted in the consumer, which reduces the usefulness of the design pattern.

Among the systems studied, only SafeJava [4] and Universe Types [20] discuss this. Generic Universe Types does not currently support this; however, it is mentioned that it would be a future extension to the system [18].

4.2 Other Features

Some systems we examined contained unique features not found in others:

- **Generics**: Only Generic Universe Types discussed generics at length. Existing programs would be likely to use generics extensively, especially since Java's runtime library has had generics for a long time now.
- **Inheritance**: Only Clarke's Ownership Types was silent about inheritance. This is an important feature to have, because existing Object-Oriented programs leverage inheritance extensively to structure large programs. The lack of inheritance would impede a system's ability to be used immediately.
- **Ownership Parameters**: Only the Universe Types system did not support ownership parameters. This is a good feature to have because it allows context-sensitive ownership information to be expressed in the program. While we were not able to construct an example that demonstrated this as a weakness, we conjecture that this would be a handicap.
- **Ownership Constraints**: Only the SafeJava and Multiple Ownership systems allowed constraints to be placed on owners. This is not strictly necessary to be able to correctly express the intention of the programmer; however, it would be useful in verifying the correctness of programs.

4.3 Program Analysis Benefits

Programming languages integrated with an ownership type system can provide a number of benefits for program analysis. Specifically, in terms of effects, alias control and local reasoning.

4.3.1 *Effect Specifications*

SafeJava, Universe Types, and Multiple Ownership include effect specifications in some form. Effect specifications are useful because they allow analysis tools to make stronger assumptions about programs, increasing the accuracy of analyses.

Universe Types gives the simplest facility for effect specification: the `pure` and `impure` method annotations to indicate the presence of side effects. This is used to constrain methods that may be invoked on objects with the given ownership properties: impure methods are not permitted in contexts where only read access is available.

SafeJava provides finer-grained control, using `reads` and `writes` clauses listing individual members of a class that a method operates on. This is used to the same effect as in Universe Types: restricting the possible method calls that can be made on objects based on their ownership properties.

Multiple Ownership uses `intersects` and `disjoint` clauses, which express the relationships between ownership contexts, to give a conservative, sound, *must* analysis of the areas of the heap written to or read by an expression. This information is also used to verify that methods do not violate their encapsulation discipline. Information on the effects of methods is assumed to be present, therefore in practice it requires some means of distinguishing pure methods from impure ones.

4.3.2 *Alias Control*

All the systems impose an encapsulation discipline. This is useful in controlling where in the program aliases for values are allowed to exist.

If the **owners-as-dominators** discipline is used, then aliases are allowable only in places where the object context is a child of the object being referenced. If the **owners-as-modifiers** discipline is used, then aliases are allowed anywhere in the program. This is an explicit design choice by Dietl et al. [20].

In both situations, Ownership Types do not assist in alias analysis beyond the fact that aliases are restricted to a portion of the program. However, this property – local reasoning – is useful in the general case, and is discussed in the following section.

4.3.3 *Local Reasoning*

The benefits to local reasoning are dependent on the encapsulation discipline that the system enforces. We find that **owners-as-dominators** tend to allow for analyses with more local reasoning owing to its strictness and restriction as to where variables can be used, depending on its type. **Owners-as-modifiers** relaxes the constraint significantly, and would not be as beneficial for automated program analysis tools.[7]

Having additional annotations does allow some extra information to be captured by the system, ameliorating the deficiencies in using the **owners-as-modifiers** discipline for local reasoning. Still, it does not bring the level of encapsulation that **owners-as-dominators** provides.

Nonetheless, the expressivity concerns presented in section 4.1 are more important when deciding which system to use and which encapsulation discipline to follow. We find that as a general trend,

[7]In spite of this, if we modify the definition of local reasoning to also include the *programmer*, then **owners-as-modifiers** does assist the programmer is comprehending what a component does. Alas, this does not really help program analysis tools.

Table 1: Summary of Expressivity and (Analytical) Benefits of Ownership Type Systems

● Supported
○ Partially Supported
× Not Supported
△ Not Discussed

	Immutable View	Mutable View	Binary Operations	Ownership Transfer	Generic Types	Class Inheritance	Ownership Parameters	Ownership Constraints	Multiple Owners	**Alias Control**	**Effect Specifications**	**Local Reasoning**
Ownership Types [16]	×	×	×	×	△	△	●	×	×	●	△	●
SafeJava [4]	●	●	×	●	△	●	●	●	×	●	●	●
Universe Types [17]	●	×	●	●	△	●	×	×	×	×	○*	●
Generic Universe Types [21]	●	×	●	△	●	●	●	×	×	×	○*	●
Multiple Ownership [8]	●	●	●	△	△	●	●	●	●	●	●	○†

* UT/GUT only supports `pure` and `impure`, rather than the more general `reads` and `writes` declarations.
† The ownership wildcard `?` limits the local reasoning possible.

expressivity of programs increase as the permissivity of the encapsulation discipline decreases.[8] It is pointless to adopt an encapsulation discipline that does not allow a program to be written, because not only would there not be any functional program, there would be no need to analyse the program.

A summary of our findings is listed in table 1.

5. CONCLUSION

Our exploration of the various type systems here surfaced a number of issues and caveats which may limit their use or hinder mainstream adoption.

For one, the type systems presented are all pure, static type systems, but the language that they target, Java, does not have such a type system. Notably Java has runtime polymorphism, reflective features, and relies on dynamic casts to cope with certain situations (for example, casting a collection of abstract objects to subtypes for use). As such, in practice compile-time checking alone is insufficient [4] and runtime checks are required, especially if Java interoperability is a requirement.

Annotations are also a burden on programmers. Ownership inference [4] is possible to an extent but cannot be done in general, as there are complex ownership scenarios (iterators being the prime example) that cannot possibly be inferred from source code. The implication is that programmers will need to provide some amount of annotations. This amount varies with the various systems, but adds significant cognitive burden due to the sheer number of new constructs [4] or heavily utilise concepts which are not immediately evident from source code [8]. It is also easy to misuse annotations: for example, trivially letting ownership checks pass by declaring everything as owned by `world`.

In summary, while benefits of ownership types are undeniable and they provide an unprecedented amount of safety and encapsulation, they can also be cumbersome to work with. We postulate that they will see most use in safety-critical applications, where their strengths in ensuring the correctness of programs can outweigh the costs of using them.

[8]The trivial case being no restrictions on expressivity when there is no limit on encapsulation.

6. REFERENCES

[1] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *Proceedings of ECOOP*, 2004.

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

[3] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In *Proceedings of ECOOP*, 1997.

[4] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proocedings of OOPSLA*, 2002.

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

[7] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of PLDI*, 2003.

[8] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *Proceedings of OOPSLA*, 2007.

[9] N. Cameron and J. Noble. Encoding Ownership Types in Java. In *Proceedings of TOOLS*, 2010.

[10] N. Cameron and J. Noble. Generic Universes Compiler, Jan 2010. `https://ecs.victoria.ac.nz/Main/Encoding`.

[11] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of NSW, 2003.

[12] D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Proceedings of OOPSLA*, 2002.

[13] D. Clarke, J. Noble, and J. Potter. Simple Ownership Types for Object Containment. In *Proceedings of ECOOP*, 2001.

[14] D. Clarke, J. Noble, and T. Wrigstad. *Aliasing in Object-Oriented Programming: Types, Analysis, and Verification*. Springer-Verlag, 2013.

[15] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming: Types, Analysis, and Verification*, pages 15–58. Springer-Verlag, 2013.

[16] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of OOPSLA*, 1998.

[17] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects*, pages 72–112. Springer-Verlag, 2008.

[18] W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, ETH Zurich, 2009.

[19] W. Dietl. Generic Universe Types Checker/Inference, Jan 2014. `https://ece.uwaterloo.ca/~wdietl/inference/`.

[20] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *Proceedings of FOOL/WOOD*, 2007.

[21] W. Dietl, S. Drossopoulou, and P. Müller. Separating Ownership Topology and Encapsulation with Generic Universe Types. *ACM Transactions on Programming Languages and Systems*, 33(6):20:1–20:62, 2011.

[22] W. Dietl, M. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *Proceedings of ECOOP*, 2011.

[23] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[24] W. Dietl and P. Müller. Ownership Type Systems and Dependent Classes. In *Proceedings of FOOL*, 2008.

[25] W. Dietl and P. Müller. Object Ownership in Program Verification. In *Aliasing in Object-Oriented Programming: Types, Analysis, and Verification*, pages 289–318. Springer-Verlag, 2013.

[26] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of OOPSLA*, 1991.

[27] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Springer-Verlag, 2002.

[28] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, pages 131 – 140. Fernuniversität Hagen, 1999. Technical Report 263.

[29] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *Proceedings of OOPSLA*, 2007.

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

[31] Oracle Corporation. Iterator (Java Platform SE 8), April 2015. `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 8th edition, 2008.