Regions as Owners [*]

A Discussion on Ownership-based Effects in Practice

Johan Östlund Tobias Wrigstad

Uppsala University

{first.last}@it.uu.se

Abstract

A number of proposals in the literature combine ownership notions and computational effects. The combination is sensible: ownership gives strong guarantees about disjointness of subheaps and therefore lends itself well to modular reasoning about effects.

We argue that these proposals, while formally appealing, turn out to be of little use in practice. The problem lies in the absence of owners that are provably disjoint in actual programs, leading to many situations that are safe in practice, but cannot be proven so by the type system. In this short note, we show a number of examples that cannot be expressed using the studied systems, and analyze the underlying cause. We also propose a region system that avoids these problems by unifying owners and regions in a natural way that furthermore allows expressing effects more precisely than previous ownership-based effect systems.

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

General Terms Object-Oriented Programming, Aliasing, Confinement

Keywords Object-oriented programming, Ownership Types, Regions, Effects

1. Introduction

Ownership types gives a structure to the object graph by imposing a nesting relation on objects and encapsulating objects in their owning aggregates. In classic, Clarkean ownership [7, 12] each object is owned by another object (or the special owner **world**) and can be the owner of other objects. In the standard terminology, each object defines a single *context*—its representation—usually called **this** or **rep**. A context is an isolated subheap, and objects owned by a context are fully encapsulated inside its defining object. The containment invariant of Clarkean ownership systems is the so-called *owners-as-dominators* property; any path from a root in the system to an object contains the object's owner. As a consequence, the heap is hierarchically shaped.

Systems that support owners-as-dominators are generally called *deep ownership* systems. In these systems, the ownership relation forms a tree rooted in the outermost context **world** which all objects are (transitively) inside. In deep ownership systems, references that point from an enclosed context to an enclosing one are permitted, but not the converse.

For static checking, ownership types systems capture the nesting structure in objects' types. In addition to its owning context, a type can be parameterized with *permissions* to reference objects in other contexts. The key to deep ownership is to require that all permissions of an object denote contexts *outside* its owner. Shallow ownership [2] relaxes this property in an unstructured fashion, with unclear confinement guarantees as the result.

Confining aliasing is one approach of several to manage alias problems in object-oriented programs [16]. Sometimes, controlling the computational effects incurred via aliases are a more viable approach than overly restricting a program's object graph. Usually effects are computed by extending the type rules to propagate information about what data is touched by a particular expression. Effect information facilitates reasoning about whether two expressions are disjoint, *i.e.,* only touch non-overlapping data. Disjointness of effects may be very useful to determine if a program transformation is safe to perform, or to decide whether two computations may be run safely in parallel, especially important now in the advent of ubiquitous multicore chips. However, managing aliasing and managing their effects go hand in hand, as several researchers have noted [8, 13]; the possibility of aliasing severely impacts the possibilities to guarantee effect disjointness. To deal with this problem a number of proposals have combined effects with ownership types or uniqueness, which both restrict aliasing in different ways.

In this paper we survey a number of existing systems which fall into the description above. Joe_1 combines deep ownership and effects [8], OOFX uses abstract regions and effects [13], MOJO is a language with multiple ownership and effects [6], ODE combines ownership domains and effects [20] and DPJ uses regions and effects for deterministic parallelism [3]. All these systems have their respective strengths and weaknesses, as this survey will show. Last, we propose a novel system combining owners and regions to address some of the identified problems.

Contributions Our contributions in this paper include a survey of existing systems with effects for object-oriented languages showing their respective pros and cons, insight into the discrepancy between theory and practice, and a proposed solution to some of the identified problems.

2. Object-Oriented Effects Systems: A Survey

In this section we overview a number of object-oriented effects systems from the literature and identify a number of problems that render these less useful in practice than one, perhaps, would expect.

2.1 Running Litmus Tests

Where it makes sense, we use a number of recurring examples in our survey. The examples are chosen as examples of commonly occurring patterns with respect to ownership structure of programs. In honor of the canonical list example for ownership types systems, we use list-based examples, but they should nonetheless be easy to generalize. The tests are:

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

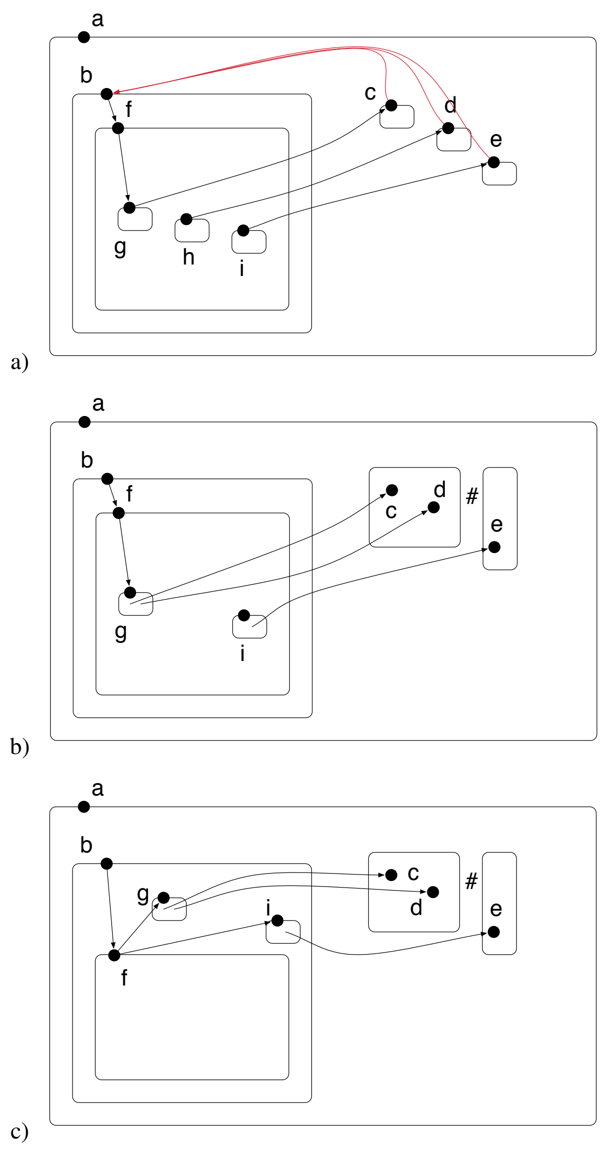


Figure 1. Ownership graphs corresponding to our three scenarios. Arrows denote references and red arrows denote *absence* of references.

1. An object with a collection of external objects disjoint from the set of enclosing objects—can this be expressed and can effects on the list objects be distinguished from effects on **`this`**?
2. An object with two collections of external objects in its rep—can effects on the two collections be distinguished?
3. An object with two collections of external objects where the collections are also external to the object–can effects on the two collections be distinguished?

The first example deals with *vertical disjointness*, the second with *horizontal disjointness*, and the last whether disjointness information may be transferred in a modular fashion. Fig. 1 shows the ownership graphs we assume as the setting for these examples. In addition to looking at how each system deals with this common set of situations we also sometimes comment on other pros and cons regarding the different system designs.

Example 1: External, but not Enclosing Fig. 1 a) shows a, generally frequently occurring situation in practice. Let f be a list with links g, h, i referencing external objects c, d, e. This example investigates the ability to express outgoing aliases and the ability to distinguish between the current and the enclosing context. Ideally, modifying the objects c, d, e should not count as a modification to the list itself.

Example 2: Same Context, but Disjoint In this example, illustraded in Fig. 1 b), we consider two lists (g, i) both referencing external objects in the same context (c, d, e). This example investigates the ability to reason about disjointness of different subobjects in the same representation.

Example 3: External, but Disjoint In this example, depicted in Fig. 1 c), we consider two lists both referencing external objects in the same context, but now the lists themselves belong to external contexts. The rationale for this example is to investigate whether disjointness information can be transferred in a modular fashion.

We now proceed with surveying object-oriented effects systems from the literature.

Effective Ownership Systems Before we start looking at the systems of the survey we wish to acknowledge some of Lu and Potter's work on expressing ownership in terms of effects. With *Effective Ownership Types* [18] Lu and Potter show how to use effects to achieve an ownership discipline similar to *owners-as-modifiers* of Universes [19]; an object may be referenced from any context, but an update must go through the object's owner. Another work, similar in spirit, is *Variant Ownership Types* [17], where the type system separates permission to reference an object from permission to update it. The result is a flexible system which provides a property similar to owners-as-modifiers, but do not allow any reasoning about effect disjointness as the other systems in this survey do.

Both these systems are interesting and flexible ways of expressing encapsulation disciplines. However, effects in these systems are not used to reason about disjointness, but to enforce encapsulation. A different treatment of read and write effects might allow this, but as this is not addressed in their work it is out of scope for this survey.

2.2 Joe$_1$ and The Disjointness of Type and Effect

Almost 10 years ago, Clarke and Drossopoulou proposed the novel formal language Joe$_1$ [8] which married a deep ownership type system with an effects system to show the correlation between types and effects. In particular, they showed how by just looking at whether two types contain different owners, one can tell whether they can be aliases. Deep ownership ensures that the representation of an object is fully encapsulated inside it. This fact, along with the observation about aliasing gives a notion of disjoint aggregates; entire subheaps which are guaranteed to be non-overlapping. Of course, when computing effects such information is very useful. In Joe$_1$ one may advertise effects for a particular strand of a subheap (*e.g.,* the owner p or two levels of nesting below p, $p.2$) or an entire subheap rooted at some owner p, `under`(p). The $p.n$ effect is particularly expressive, and allows *e.g.,* expressing disjointness between swapping the compartments of a tuple and the contents of the compartments, etc.

Formally, disjointness of types in Joe$_1$ is very straightforward: two contexts p and q are disjoint, written $p \# q$, if $p \prec^+ q$, meaning that q is a context containing p, but not equal to p. Two types are disjoint if they are not related by subclassing or if a pair-wise comparison of the owners occurring in the types shows at least one pair that is disjoint. If the variables x and y have disjoint types, then x and y cannot be aliases, and since objects' representations are fully encapsulated in deep ownership systems, any effect to the object pointed to by x or its representation will be disjoint from an effect to y.

Effects on External Contexts We will now survey how Joe$_1$ fairs in the context of the three problems previously formulated. For this sake we define a linked list class thus (stolen from the presentation of Joe$_1$ [8]):

```
class Link<owner,data> {
  Link<owner,data> next;
  Data<data> data;
  Link(Link<owner,data> n, Data<data> d) writes this {
    this.next = n;
    this.data = d;
  }
}

class List<owner,data> {
  Link<this,data> head;
  void add(Data<data> d) writes under(this) {
    head = new Link<this,data>(head, d);
  }
}
```

The class `List` has one context parameter (`data`) in addition to its owner. The links in the list are pointed to by the `head` field which is owned by the list representation context and thus fully encapsulated within the list instance. Each link has a `next` field which is owned by the link's owner, that is the list instance representation, and a `data` field owned by the `data` context.

The situation we are interested in studying is depicted in Fig. 1 a), and the question is whether the system can tell that an effect on, `c` will not affect object `b`? Inside object `b` there is no way this can be done, since we cannot distinguish the owner of `c`, `d` and `e` from the owner of `b` (which in this case are the same.) However, from the point of view of object `a` Joe$_1$ can distinguish effects internal to `c` from effects on `b`, but if a method call on `c` is made which has an effect on **`owner`**, then such an effect cannot be distinguished from an effect on `b`. This is similar to the next situation we shall consider.

No Support for Horizontal Disjointness The second problem formulated is illustrated by the following code:

```
List<this, elem> list1;
List<this, elem> list2;
```

Both lists belong to the representation of some object and they both store elements from the same external context. There is no way to tell whether these lists are different lists or aliased. It is quite reasonable to want such a setup, for example to access a set of elements in different precomputed orders. The point is that if we know that they are different there is no reason to prevent iterating over one of the lists while sorting the other. Systems such as OOFX [13], DPJ [3] and Ownership Domains [1] support partitioning an object's representation into disjoint *regions*, which avoids this problem.

Little Vertical Disjointness in Practice A likely scenario in which such a list may be used is this (also from the Joe$_1$ paper), this corresponds to the third test:

```
List<p, world> list1;
List<q, world> list2;
```

Assuming $p \# q$ we can easily deduce from the rules for disjoint types that `List<p, world>` # `List<q, world>`, *i.e.,* that `list1` and `list2` must point to different lists. While this is both sound and great, we argue that this is exactly where Joe$_1$ runs into problems: usually one cannot determine $p \# q$. The only contexts, in general, that we know are disjoint are **`this`** # **`owner`**, **`this`** # **`world`** and **`this`** # **`p`**, assuming **`p`** is a context parameter. The problem is that other than **`this`**, **`owner`** and **`world`** the only contexts known are those declared as context parameters. The only known fact about these parameters is that all of them are outside **`owner`**, meaning that they include the **`owner`** context, but they may very well be equal to **`owner`**. A solution to this problem is to include the relations between context parameters in the class header, giving a class header looking thus:

```
class C<owner,
        p strictlyOutside owner,
        q strictlyOutside p> { ... }
```

While this indeed solves the problem of transfer of context relation information it is inflexible. Inside class `C` we would be able to deduce `List<p, world>` # `List<q, world>`, but this comes at a cost. Annotating classes this way obviously propagates the problem to the user of the class. For such a type to be valid it has to be instantiated with owners that are provably strictly outside each other, which means such a type can only be used in classes which themselves have access to such relation information. Another problem with this strategy is that there may have to be (at least) two versions of many classes in the system, one that allows contexts to overlap and one that does not, which is hardly desirable.

So, to conclude, the practical usefulness of Joe$_1$ is hampered by the fact that there simply are not enough owners around and there is not enough information about their relations to express disjointness (in most situations).

2.3 Multiple Ownership and Disjointness

Cameron *et al.* propose MOJO [6], a language with multiple ownership and effects. In MOJO contexts are called *boxes* and objects owned by a box is said to be in that box. In contrast to classic ownership types, where every object belongs to a single context which is fixed for life, MOJO allows an object to be in multiple boxes at the same time. Such an object is said to belong to the *intersection* of these boxes. Multiple ownership is not compatible with the strong notion of encapsulation given by owners-as-dominators. On the other hand, multiple ownership can express additional programming patterns. In the paper Cameron *et al.* use as running example a company with projects, tasks and workers. In a traditional ownership system tasks and workers must belong to a single project (owners-as-dominators). Modeling the quite reasonable situation that a worker is involved in several projects is not possible, unless everything is owned by the same context, say the company, which gives a flat and unstructured ownership graph, without clear gains over using a language with no ownership. MOJO, however, with its support for multiple ownership can model such a situation quite nicely.

External but not Enclosing MOJO supports disjointness-annotations which guarantee disjointness of contexts. Using this feature it is possible to prove disjointness at any level. However, using such annotations comes at a cost, which we will discuss shortly.

Split Representation MOJO suffers in the same way that Joe$_1$ does when it comes to our second litmus test. MOJO does not support splitting of the representation context and can therefore not handle this situation in a satisfactory way.

Explicit Disjointness is Expressive but Inflexible MOJO sports an effects system for multiple ownership. The effects system is a straightforward and unsurprising extension of Joe$_1$, but the fact that boxes may intersect calls for extra machinery. MOJO classes can declare that two boxes *intersect* or are *disjoint*, which governs the way boxes may be combined. Declared disjoint boxes may not share objects, and thus effects on such boxes are disjoint.

```
class C<owner,a,b,data> where b disjoint a {
  ...
  List<a, data> list1;
  List<b, data> list2;
}
```

In a class such as C above, context parameters a and b are declared to be disjoint. Upon instantiation of such a class the type system ensures that only provably disjoint contexts may be used as actual context parameters. Given this declaration we know that the two lists reside in different boxes and thus cannot be aliases. This is clearly an improvement over Joe_1, but it comes at the cost of inflexibility. The problem is essentially the same as described in the discussion about Joe_1. Several versions of the same classes may have to exist in the system. On the other hand disjointness in MOJO is somewhat easier to achieve than strictly outside in Joe_1, since two boxes in the same local context can be declared disjoint, whereas strictly outside needs to be established by a client of the class.

An aside–Disjointness and Aliasing MOJO allows a programmer to use final variables as owners (example taken from [6]):

```
final Worker<this> w1 = new Worker<this>;
final Worker<this> w2 = new Worker<this>;
w1 disjoint w2;

w1.delay(); // EFF: w1 / w1
w2.delay(); // EFF: w2 / w2
```

In the example two workers are created and stored in final variables. The variables are declared to be disjoint, which means that the effects of calling delay on the workers are disjoint (the EFF-comments show the read and write effects of calling the methods, respectively.) However, this is only true if w1 and w2 are not aliases. In this example we can trivially see that this is the case, but in general the type system cannot know this without some other machinery to track or constrain aliasing. Not knowing that w1 and w2 are unaliased means that we need to be conservative and disallow a disjointness declaration. It is unclear to us what the impact is in practice.

2.4 Ownership Domains

Ownership Domains [1] allows a programmer to specify arbitrarily many disjoint contexts (domains) in a class. The key idea of ownership domains is that the referencing policy is not fixed by the system but specified by the programmer by explicitly linking domains—"define, not confine". In his dissertation, Smith extends Ownership Domains with effects [20]. Declaring multiple disjoint contexts allows splitting the representation of an object into provably disjoint parts which enables more precise effects.

Split Representation Litmus test one, formulated above and depicted in Fig. 1 a), is quite easily achieved in Ownership Domains. We would like to distinguish effects on object c from effects on object b. By declaring two disjoint domains in object a and putting object b and the list elements in different domains effects on the list elements are disjoint from effects on object b. We can also handle Test 2 in a similar fashion.

External Disjointness In Ownership Domains one may annotate a class with assumptions about how domains passed as parameters may reference each other. However, these annotations can only express that a particular region must be able to reference another region, not the converse. Therefore we cannot, in general, know that two external domains are distinct and so the third test, where we wish to distinguish two external lists is not easily solvable in Ownership Domains.

The flexible nature of ownership domains does not give any aliasing guarantees for objects in domains and there is no *deep* notion of ownership. Nothing prevents an object in a supposedly internal subdomain, with full access to its enclosing objects, to be arbitrarily exposed, which makes effects tricky to track.

As a consequence, it is not surprising that ownership domains can express many situations, but it is unclear whether an automatic system (or a human, for that matter, when non-trivial code bases are used) will be able to track effects in ODE, let alone determine that the flexibility is not used in such a way that encapsulation is effectively voided.

2.5 Effects for Deterministic Parallelism

Deterministic Parallel Java (DPJ) [3] uses regions and effects to reason about the deterministic execution of parallel code. A class may declare any number of regions and also declare each of its fields to belong to a particular region. This gives a partitioning of objects' representations. Further, regions may be passed as parameters to types, similar to the other systems in this survey. The following binary tree class is used as an example in the DPJ paper:

```
class TreeNode<region P> {
  region Links, L, R, M, F;
  double mass in P:M;
  double force in P:F;
  TreeNode<P:L> left in Links;
  TreeNode<P:R> right in Links;
  TreeNode<*> link in Links;

  void computeForces() reads Links, *:M writes P:*:F {
    cobegin {
      /* reads *:M writes P:F */
      this.force = (this.mass * link.mass) * R_GRAV;
      /* reads Links, *:M writes P:L:*:F */
      if (left != null) left.computeForces();
      /* reads Links, *:M writes P:R:*:F */
      if (right != null) right.computeForces();
    }
  }
}
```

The class declares one region parameter P, five regions Links, L, R, M and F. RPLs, short for region path lists, are colon-separated lists of region names, and used to express region nesting. Region nesting forms a tree structure, similar to ownership. One RPL is nested inside another RPL if the former is a prefix of the latter. RPLs are crucial when determining effect disjointness. Two RPLs, say P:M and P:F are disjoint because the names of the regions they describe differ. RPLs may be determined to be disjoint by differing either from the left of from the right. We will not dig too deep into the details of how to determine disjointness. Instead we concentrate on some of the aspects of the example above. The cobegin block runs each of its statements in parallel. It can do so safely because the effects of the statements can be determined not to interfere. The two calls to the computeForces method affect the left and right tree, which are disjoint (they differ in the L and R position P:L:*:F and P:R:*:F.) As we can see disjointness of regions (and effects) is closely tied to the sequences of region names in the RPLs. This means that a programmer must know the nesting structure of regions. This is not a problem for recursive data structures, such as the binary tree here, but in general abstraction is severely impacted by this scheme. It should also be noted that region nesting and disjointness is achieved by linearity (uniqueness), or rather by keeping the data owned by **rep** in ownership terminology. The left and right branches are disjoint because by well-formed construction of the tree this must be the case. However, the example above would suffer from storing other data than primitive data in the tree, unless it was copied (into the node's **rep**), which is expensive, because potential aliasing would either void guarantees or limit the parallel execution possible.

2.6 Effects on Abstract Regions

Greenhouse and Boyland [13] propose the language OOFX which has an effects system expressed in terms of abstract regions in a class's public interface. Fields are mapped to regions and effects on a field are advertised in terms of its defining region. In terms of object-orientation, OOFX offers good abstraction capabilities in that only regions have to be visible in the public interface of a class, not the names of particular fields. Regions may be arbitrarily extended in subclasses, but method overriding must preserve effects.

OOFX's Regions are of a Shallow Nature Unless care is taken, fields of different regions could be aliases, which would cause an effect to one region to also be an effect on the other. Correctly handling such situations requires tracking aliasing between regions which is very hard, and OOFX instead attempts to handle this problem by employing a notion of uniqueness, called *unshared fields*, to avoid aliasing altogether. Sadly, programming with unique variables causes a wealth of problems on its own [4, 5] due to the problems of maintaining uniqueness. Further, plain uniqueness is a shallow property unless full encapsulation such as Islands [15] is used, which causes additional pain. *External Uniqueness* [9, 21] solves some of the problems, but needs deep ownership (or some similar heavy machinery [14]) to be realized, and the introduction of deep ownership here would lessen the need for uniqueness in the first place.

OOFX's regions can express regions for unique nodes in a linked list (an example can be found here [13]). This precludes doubly-linked lists, but trivially allow distinguishing direct effects on objects in the list since these are uniquely referenced. In a doubly linked list, disjointness cannot be proven without additional heavy machinery such as alias analysis, as far as our understanding goes.

2.7 Discussion

We have now presented our survey of existing systems looking at some of their strengths and weaknesses. In summary we can conclude that disjointness of effects is hard to achieve and track in practice. In Joe_1 the problem is mostly the shortage of contexts which are provably disjoint, which in turn is a consequence of the lack of transfer of disjointness information and single representation. MOJO solves this by annotating classes with a `where`-clause specifying the disjointness of boxes. While this indeed works for transfer of disjointness information it severely impacts reusability and may call for several implementations of the same class. OOFX's regions offer a way of splitting the representation of an object into several disjoint parts. This is very appealing because it allows reasoning about disjoint modifications in the representation of a single object. OOFX suffers from aliasing problems which may be overcome by brittle alias analysis or very restrictive unshared fields. Ownership Domains offers loads of flexibility in that the reference scheme is not fixed by the system but specified by the programmer. This allows the programmer to express many programs not easily realizable in traditional ownership systems. However, the flexible nature of ownership domains make tracking effects difficult since there is no such thing as a dominating node or equivalent that bounds from where effects may be incurred. As a consequence, how useful ODE is in practice for safe parallelization and effective reasoning is unclear to us. Deterministic Parallel Java uses region path lists to calculate disjointness of effects. To establish disjointness, information about the nesting of regions in a class must be exported in a way that impacts abstraction negatively.

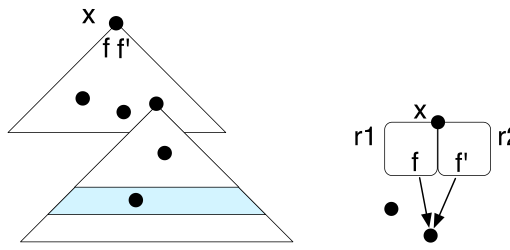


Figure 2. Joe_1 (left) and OOFX (right). In Joe_1, each object defines a single subheap but it is possible to talk about effects on different strands of the subheap (*e.g.,* the teal-colored stripe). In OOFX, regions are shallow and encapsulate fields only, not the contents of the fields which may be shared (with the exception of unshared fields).

Drawing on the conclusions of this survey we sketch next a minor change with major impact. Our proposal combines regions and owners to address some of the issues identified in the previous systems.

3. Unifying Regions and Ownership in Joelle

Clarke and Drossopoulou's effect shapes are *deep* and handle aliasing elegantly by basing the system on deep ownership types. In contrast, Greenhouse and Boyland's effect shapes are *shallow*, unless regions contain only unshared fields. In practice, Clarke and Drossopoulou's system is likely, we argue, to be of little value due to the lack of disjoint owners. In some cases, disjointness of types can be enough for effect disjointness, but as soon as a method on an object has an effect on its owning context, disjointness of type means little. Greenhouse and Boyland's system seem to be of little practical value, due to its being based on a complicated and brittle notion of uniqueness to battle the effects of aliasing. Fig. 2 shows this pictorially.

We propose a simple region construct as a solution to the problems of both these systems that unify OOFX-style regions with Clarkean owners in a natural and straightforward fashion[1]. Our notion of region is a refinement of a context; a region is a carved-out subset of a context subheap including fields with outgoing references. A class may define any number of *disjoint* regions that can act as owners of objects. Further, we map each field into a single region to represent the slice of the top-level object that holds the field. Fig. 3 shows this pictorially.

If an owner is interpreted as the set of objects inside its transitive closure, then if a class defines two regions r_1 and r_2, $\texttt{this} = r_1 \cup r_2$ and $r_1 \cap r_2 = \emptyset$.

Just like in OOFX, regions are part of a class' public interface, and writing a field in a region r is a write effect on r. Similarly, a write effect on an object owned by r (nested, perhaps deeply, inside the region) is also a write on r. For concreteness, here is an example of our surface syntax:

```
class BinaryTree<owner, data> {
  region left, right;
  BinaryTree<left, data> left in left;
  BinaryTree<right, data> right in right;
```

[1] It should be noted that in the related work section of [3] it is observed that ownership normally has a one-to-one mapping between regions and objects.

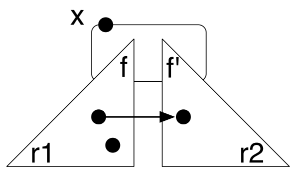


Figure 3. Regions as Owners in Joelle.

```
  Object<data> value;

  boolean contains(Object<data> obj) reads(this) {
    return value == obj ||
               findInLeft(obj) ||
                   findInRight(obj);
  }

  boolean findInLeft(Object<data> obj) reads(left) {
    return left.contains(obj);
  }

  boolean findInRight(Object<data> obj) reads(right) {
    return right.contains(obj);
  }
}
```

Since all regions in a class are refinements of a single context, objects belonging to different regions are effectively siblings, and it makes sense, also from a programmer's point of view, to allow all regions of a class to refer to each other's objects. Nothing prevents allowing regions to be additionally refined, equivalent with the hierarchical nesting of regions in OOFX, although we do not consider that here.

```
class BinaryTree<owner, data> {
  region left, right, shallow;
  BinaryTree<left, data> left in shallow;
  BinaryTree<right, data> right in shallow;
  ...
```

In the above example, the binary tree class defines a third region `shallow`, that only contains the fields `left` and `right`; the equi-named regions encapsulate the left and right branches of the tree. A reason for using a separate region for holding the fields could be *e.g.,* to facilitate reasoning about writes to the top-level fields that do not effect any nested state. In this particular care, an effect on `shallow` is equivalent to `this.1` in Joe$_1$.

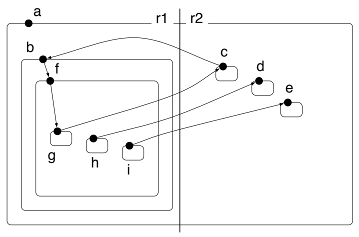
If an object defines a single region r, this region is effectively the same as the standard `this` region. Notably, if an object defines a region r and only fields in r use r to form types, then the fields in r dominate the objects in the region. This is, in a sense, a generalization of *external uniqueness* [9, 21]; if r in our example has a single field, then the reference in that field is a dominating edge to all objects (transitively) in r. A big advantage of regions over uniqueness is avoiding having to deal with the inherent slipperiness of unique values [4].

Constructing a type system to go along with the above design is straightforward.

3.1 Applying Joelle Regions to Our Examples

In this section, we examine the examples from previous sections to see how such a simple idea as splitting representation into several disjoint parts helps overcoming the aforementioned problems.

External, but not Enclosing The key to expressing this situation in Joelle is to divide the context in a into two disjoint parts, r_1 and r_2 say



and use r_1 as the owner of b, and its transitive state, and r_2 for the objects c, d, e. As a consequence, a can distinguish between the object containing the list (b) and its data objects, but b cannot, nor can the list (f). See the discussion on "External, but Disjoint".

Notably, our system cannot express that c may not reference b if b and c should belong to the same context. However, this should pose no problem since effects to the different objects will be reported as effects to different contexts, so any method in c that touches b will report this clearly and the disjointness is clear in the context of a.

Same Context, but Disjoint This is trivially expressed by context refinement. Declare two regions as disjoint subsets of the current context and place the lists in separate regions.

```
class Client<data> {
  region a, b;
  List<a,data> g in a;
  List<b,data> h in b;
}
```

External, but Disjoint Expressing that two outgoing references point into two external, disjoint aggregates is not possible in our system without allowing statements of disjointness on context parameters á la MOJO. Such statements have a side-effect of constraining reuse of classes in situations where there is no disjointness and should be used sparingly, only for data structures where disjointness is a key property of sound behavior.

4. Concluding Remarks

In this short note, we have discussed ownership-based effects systems in the context of a couple of examples that examine issues with vertical and horizontal context disjointness. We have also showed on-going work on an amalgamation of OOFX-style regions with Joe$_1$-style ownership-based effect shapes (reminiscent of DPJ) to overcome limitations of both these systems and improve the precision of effects. Our regions are being implemented and tested in the context of our Joelle language for parallel programming [10, 11], and results of practical evaluation will be fed back into the design to further refine our proposal. We are also working to solve the problem of transferring disjointness information in a modular way. We are confident that additional experiments with rewriting Java-like programs in Joelle will increase our understanding of the practical implications of deep ownership and effects systems based on ownership, not only for the usability of ownership-based effects systems in practice, but hopefully for other ownership-related constructs as well.

References

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

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

[3] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[4] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 2–27, London, UK, UK, 2001. Springer-Verlag.

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

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

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

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

[10] D. Clarke, T. Wrigstad, J. Östlund, and E. Broch-Johnsen. Minimal ownership for active objects. Technical Report SEN-R0803, Centrum voor Viskunde en Informatica, Amsterdam, The Netherlands, June 2008. Extends [11] with a complete formalisation of Joelle's semantics, additional background and examples.

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

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

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

[14] P. Haller and M. Odersky. Capabilities for Uniqueness and Borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer, 2010.

[15] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, Nov. 1991.

[16] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[17] Y. Lu and J. Potter. On ownership and accessibility. In D. Thomas, editor, *ECOOP 2006 - Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 99–123. Springer Berlin / Heidelberg, 2006.

[18] Y. Lu and J. Potter. Protecting representation with effect encapsulation. *SIGPLAN Not.*, 41:359–371, January 2006.

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

[20] M. Smith. *A Model of Effects with an Application to Ownership Types*. PhD thesis, Imperial College, London, UK, May 2007.

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