Multiple Ownership

Nicholas R Cameron,
Sophia Drossopoulou
Department of Computing,
Imperial College London, UK
{ncameron, sd}@doc.ic.ac.uk

James Noble*
School of Mathematics, Statistics
& Computer Science,
Victoria University of Wellington,
New Zealand
kjx@mcs.vuw.ac.nz

Matthew J Smith
Department of Computing,
Imperial College London, UK
mjs198@doc.ic.ac.uk

Abstract

Existing ownership type systems require objects to have precisely one primary owner, organizing the heap into an ownership tree. Unfortunately, a tree structure is too restrictive for many programs, and prevents many common design patterns where multiple objects interact.

Multiple Ownership is an ownership type system where objects can have more than one owner, and the resulting ownership structure forms a DAG. We give a straightforward model for multiple ownership, focusing in particular on how multiple ownership can support a powerful effects system that determines when two computations interfere — in spite of the DAG structure.

We present a core programming language MOJO, Multiple Ownership for Java-like Objects, including a type and effects system, and soundness proof. In comparison to other systems, MOJO imposes absolutely no restrictions on pointers, modifications or programs' structure, but in spite of this, MOJO's effects can be used to reason about or describe programs' behaviour.

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

General Terms Languages, Theory

* This work was developed while James Noble was on leave at Imperial College London, and Microsoft Research, Cambridge.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction

> *We're tired of trees... We should stop believing in trees, roots, and radicles.*
>
> Deleuze and Guattari, **A Thousand Plateaus** [17]

In ownership systems, each object has one owner and the ownership relation forms a tree. While different versions of ownership have proved effective for a variety of tasks [2, 7, 8, 14, 18], empirical studies have shown that this ownership structure does not suit all programs [1, 6, 34, 43]. In this paper we present an ownership type system that removes this restriction and does not require the owners to be dominators, so that an object may have multiple owners, and the ownership relation forms a DAG. We make the following contributions:

- the *objects in boxes* model, a simple, straightforward model of object ownership based on sets of objects, which describes the fundamental features of single ownership, and generalises smoothly to multiple ownership.
- a language design incorporating **M**ultiple **O**wnership into a **J**ava-like language with **O**bjects (MOJO). MOJO's novel constructs include multiple ownership types, constraint declarations to indicate that two boxes either intersect or are disjoint, and a restricted form of existential ownership. Thus, existing ownership type systems can support multiple ownership via relatively small extensions.
- a formal definition for MOJO, including a type system which we have proved sound.
- an effects system for MOJO that works with multiple ownership, that again, we have proved sound.

The next section informally introduces our conceptual model of ownership, the language MOJO, and the effects system. We then give a formal presentation of the syntax, operational semantics, type and effects system of MOJO, and its soundness. The paper concludes

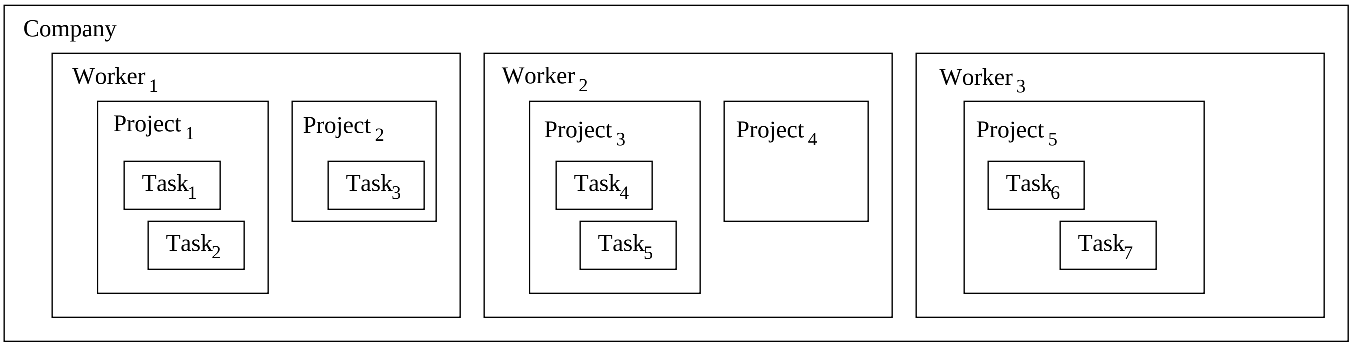

Figure 1. A Single Ownership Structure. Three `Worker`s belong to the Company, each `Worker` is working on several `Projects`, and and each `Project` has `Task`s the `Worker` must complete.

with a discussion of MOJO idioms and extensions and a brief survey of related work.

2. The Benefits of Putting Objects into Boxes

In this section we present our conceptual model — the "*objects in boxes*" model [19] — of multiple ownership and effects in object-oriented systems. We begin by modelling single ownership, then show how the objects in boxes model generalises to multiple owners. Interleaved with the conceptual presentation, we show how these models can be described using ownership types in programming languages. The examples are expressed in our core language MOJO but would apply in most languages with ownership types.

Upon reflection, given that ownership has been studied for at least ten years [38], and alias control for fifteen [25], it seems odd that only now we are presenting something as naïve as a model based purely on sets. Compared with previous work, the objects-in-boxes model focuses on ownership sets (boxes), the objects in the boxes, and the effects of computation, and abstracts away from language constructs, types, messages, capabilities, and especially the pointer structures that feature prominently in most other treatments of object ownership [2, 12, 14, 35, 37]. While some of these concerns must be reintroduced as we move from a conceptual model to a programming language, we have found the abstraction offered by the objects in boxes model to be very useful in designing and reasoning about ownership, and multiple ownership in particular. Section 5.1 discusses how features of other ownership systems can be reintroduced into our model.

2.1 Single Ownership

The object structure from figure 1 shows a company that carries out a range of different `Projects`. Each `Project` has one or more `Worker`s allocated to it, and each `Worker` has one or more `Task`s they need to complete.

The key relationship this diagram brings out is *object ownership*: each `Task` is owned by a `Project`, and each `Project` in turn is owned by the `Worker` responsible for it. Ownership models abstraction, encapsulation and aggregation: `Task`s are *part of* their `Projects`; `Worker`s are part of the Company they work for. A change to one of the parts — say a `Project` being cancelled — necessarily affects the whole abstraction in which that part is contained. Similarly, a change to a whole – say a `Worker` going on leave — may change any of its subparts — perhaps delaying all of the `Task`s comprising its `Project`.

Partitioning objects is key to ownership systems, whether they use types [14] or specifications [36]. Different systems have chosen different names for these partitions: islands [25], balloons [4], domains [2], contexts [12], regions [22] — with each name being associated with a particular detailed proposal.

We propose the neutral term **boxes** to describe these partitions: in a sense, every ownership system "puts objects into boxes" and differs in the details of those boxes. Figure 1 also gives a hint to the most fundamental semantics of these boxes: *a box is a set of objects*. So, for example, we could write $[\![\,\texttt{Worker}_2\,]\!]$ to mean all the objects contained within $\texttt{Worker}_2$'s box. Here we have:

$$\begin{array}{rcl} [\![\,\texttt{Project}_2\,]\!] & = & \{\texttt{Task}_3\} \\ [\![\,\texttt{Project}_3\,]\!] & = & \{\texttt{Task}_4, \texttt{Task}_5\} \end{array}$$

The first consequence of this model is that diagrams such as Figure 1 (which have adorned almost every ownership paper ever published) can now be ascribed clear semantics: they are just the diagrams of sets we are familiar with from primary school.

The second consequence of this model is that semantics of object composition — box nesting — follows naturally. So, for example, reading from the diagram:

$$\begin{aligned}\llbracket\, \texttt{Worker}_1 \,\rrbracket &= \{\texttt{Project}_1, \texttt{Project}_2, \texttt{Task}_1,\\ &\quad \texttt{Task}_2, \texttt{Task}_3\}\\ \llbracket\, \texttt{Worker}_3 \,\rrbracket &= \{\texttt{Project}_5, \texttt{Task}_6, \texttt{Task}_7\}\end{aligned}$$

We also have the invariant that if x is inside o, written $x \ll o$, then x belongs to the box of o. In other words:

$$x \ll o \Leftrightarrow x \in \llbracket\, o \,\rrbracket \qquad \textsc{Objects in Boxes}$$

An object's box must be a subset of its owner's box:

$$x \ll o \Rightarrow \llbracket\, x \,\rrbracket \subseteq \llbracket\, o \,\rrbracket \qquad \textsc{Box Nesting}$$

And, in single ownership, the inside relation is a tree:

$$\begin{array}{c}\llbracket\, o_1 \,\rrbracket \cap \llbracket\, o_2 \,\rrbracket \neq \emptyset\\ \Rightarrow\\ \llbracket\, o_1 \,\rrbracket \subseteq \llbracket\, o_2 \,\rrbracket \vee \llbracket\, o_2 \,\rrbracket \subseteq \llbracket\, o_1 \,\rrbracket\end{array} \qquad \textsc{Single owners}$$

These invariants should hold however we model heaps, and also independently of whether objects are permitted to change owner — type systems generally do not support ownership change; specification languages do.

2.1.1 Single Ownership Languages

In an ownership-aware programming or specification language we could define these classes as follows. First, the `Task` class contains two fields — straightforward value types giving the tasks's name and duration: the single method delays a task by increasing its duration.

```
class Task<o> {
 String name;
 int time;
 void delay() {time++;}
}
```

The `Task` class also has an *ownership parameter* `o` that is a special form of type parameter (a phantom type [24]) that records ownership information. The `Task` class needs to be ownership parametric, because different tasks will have different owners (e.g. in Figure 1, $\texttt{Task}_1$ is owned by $\texttt{Project}_1$ while $\texttt{Task}_4$ is owned by $\texttt{Project}_3$). Ownership parameters connect compile-time static types to run-time dynamic boxes. An object's owner parameter in its type represents the box it is inside:

$$x : \texttt{C<o>} \;\Rightarrow\; x \in \llbracket\, \texttt{o} \,\rrbracket \qquad \textsc{Owners as Boxes}$$

In ownership type languages, actual ownership parameters may be the formal parameters of the enclosing class (including the distinguished first parameter representing an instance's owner); "`this`" establishing that the current "`this`" instance is the owner of the new type; or final fields, establishing that the object contained in the field is the owner.

The `Project` class is also ownership parametric. `Projects` delay themselves by delaying every constituent task.

```
class Project<o> {
 TaskList<this,this> tasks;
 void delay(){
  for(var t : tasks) {t.delay();}}
}
```

The field `tasks` stores a list of the project's tasks, and is declared as `TaskList<this,this>`. This means that the list of tasks pointed to by the field, and each `Task` stored in the List, will be owned by *this particular project instance*, and therefore will be inside the box belonging to this `Project` instance, a member of the set $\llbracket\, \texttt{this} \,\rrbracket$, which will be different for each different project. The box nesting invariant ensures that an object's box is inside its owner. That is, $\texttt{this} \ll \texttt{o}$, and thus $\llbracket\, \texttt{this} \,\rrbracket \subseteq \llbracket\, \texttt{o} \,\rrbracket$.

The `Worker` class is quite straightforward, keeping a list of `Projects` owned by this `Worker` (i.e. inside its box) and delaying itself by delaying those projects.

```
class Worker<o> {
 ProjectList<this,this> projects;
 void delay(){
  for(var p : projects) {p.delay();}
 }
}
```

Consider now the `TaskList` class (the `ProjectList` class is similar) whose instances we omitted in Figure 1 for space reasons. Its implementation is rudimentary, as our focus is the ownership types involved:

```
class TaskList<o, tO> {
 Task<tO> t;
 TaskList<o,tO> next;  TaskList<o,tO> prev;

 void add(Task<tO> tt){
   if (next==nil) {
     next=new TaskList<o, tO>();
     next.t=tt;
     next.prev = this;
   }
   else {
     next.add(i);
   }
 }
 Task<tO> get(int i){
   return (i==0) ? item : next.get(i-1);
 }
}
```

`TaskList` has two ownership parameters. The first, `o`, is the "primary" owner parameter, just as in the other classes we've seen. The second, `tO`, is the ownership of the `Task` stored in each list node. In this way the ownership of the node and its contents do not have to be the same. The fields `next` and `prev` have type `TaskList<o, tO>` saying that the adjacent list entries have the same item ownership as this list entry, and the same owner as this object: all entries in a single list will be members of the *same* enclosing box; as will all the tasks — although they may be in different boxes. This differs from the fields in classes `Project` and `Worker`, which have `this` ownership, meaning that they belong to the box owned by the current object itself.

2.1.2 Effects within Single Ownership

Ownership can help determine the *effects* of a computation in terms of the objects read or written. Two computations do not *interfere* (they do not write the same objects, or do not read objects the other writes) if the the boxes involved do not intersect.

Effects systems [13, 22, 33] annotate methods with effects specifications, describing the boxes read or written. In `Task`, the fields `name` and `time` hold simple types, are local to the object, and can only be changed by the object itself. The `delay` method makes just such an assignment to `time`. The effects of, say, reading the `name` field would be `this / empty` meaning reading the "`this`" object and not writing anything. The effects of the `delay` method would be `this / this` — reading and writing the object to which the method is sent.

```
class Task<o> {  ...
   void delay()   //effect: this/this
    ... }

class Project<o> {
   TaskList<this,this> tasks;
   void delay()     //effect: this/this
     { for(var t : tasks) {t.delay();}
}
```

The `Project`'s `delay` method reads the `tasks` variable, the fields of those subordinate `Task` objects, and calls `delay` on them. From the effects of `delay()`, (reads `this`, writes `this`) we know that it will write whatever object it is called upon. The question is: which `Task` objects will be written?

Effects systems without ownership [22, 30] cannot easily distinguish *which* Task may be affected; effects like "`all.Task / all.Task`" say that `delay` on any project may read and write *any* `Task`. The upshot of this is that delaying any project must be assumed to delay *every other* project in the system.

This is precisely where boxes come to the rescue. Looking again at Figure 1, only the `Tasks` in the `Project`'s box are written. The type of these tasks, *i.e.* `Task <this>` gives that information. We interpret effects so that they apply to boxes, rather than objects: effects such as `.../this` means that a computation may write the "`this`" object itself, or any other object in its box $[\![$ `this` $]\!]$. The effects for `Project`'s `delay` method are `this / this`, so the method may read or write the object itself or any other object that it owns, but may not read or write any object outside its own box. The `Worker`'s `delay` method also has effects `this / this`.

2.2 Multiple Ownership

Single ownership requires every object to have a *single* direct owner, thus the ownership structure is a tree. While easy to understand, easy to model, and (relatively) easy to formalise and enforce, single ownership is too restrictive for many kinds of programs. Empirical studies have shown that relationships between objects and between the classes that define them are scale free networks — tangled graphs where every object is only a few hops from every other object [34, 43]. Non-hierarchial relationships cannot be modeled by trees. In [34], a study of heaps (up to 1.4 GB), found that up to 75% of ownership structures require multiple (shared) ownership, and up to 50% required "butterfly" structures. The need for multiple ownership has been independently identified in investigations of large libraries [1].

For example, imagine the following change to the `Projects`, `Workers`, and `Tasks` model in figure 2. The company has been restructured from an hierarchical style, where every project is carried out by just one worker, into a "matrix" management style where every task is assigned to *both* a project and a worker. As a result, tasks have to belong to both projects and workers; delaying a project must delay all employees who must work on tasks on that project, and similarly delaying an employee will delay all projects with which they are involved.

The topology in figure 2 cannot be described with existing ownership type systems. Classical ownership enforces a very strong owners-as-dominators policy over pointers — all paths to an object must be via its owner — so if programmers attempt to write programs describing this interconnected ownership structure, their programs will be rejected as type-incorrect. Other systems support owners-as-modifiers or effective ownership, rather than pointer control [35]; so they would at least be able to pick one of either `Projects` or `Workers` as a primary axis of organisation — say `Projects`— and grant permission to `Workers` to have pointers into tasks even though they belonged to projects. Unfortunately, when a `Worker` is delayed in such a system, it would not have permission to *modify* its `Project` objects because it does not own them.

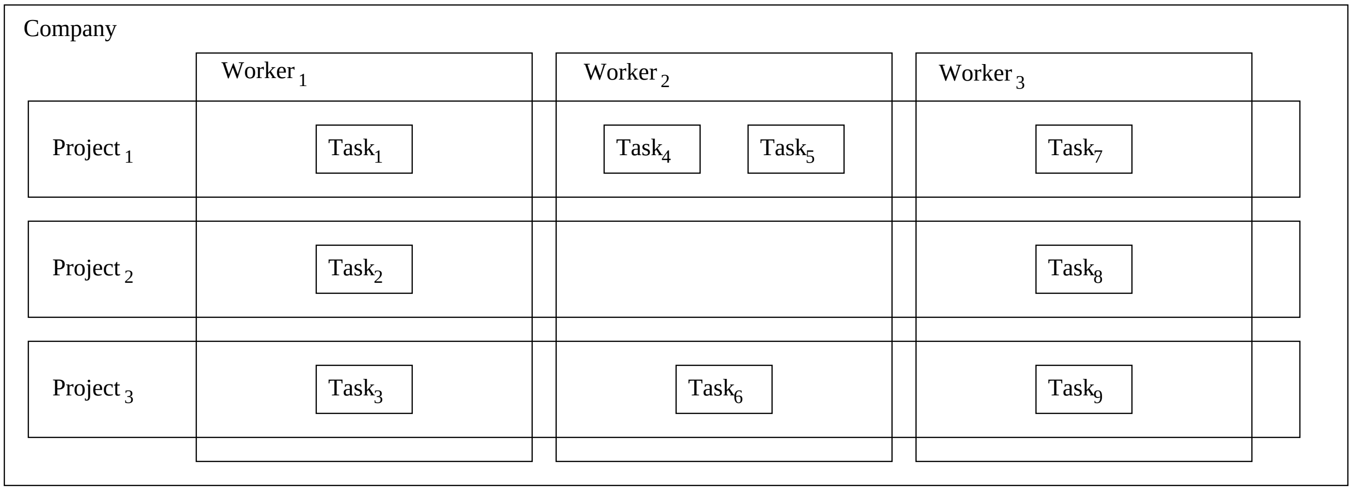

Figure 2. A Multiple Ownership Structure. The Company now requires its `Workers` to work on many different `Projects`— and different `Tasks` in a project can be carried out by different `Workers`.

In single ownership systems, programmers get around these restrictions by "flattening" or "lifting" the ownership hierarchy: rather than nesting boxes, every task, worker, and project can exist in one very large company box, and use "peer" ownership — types like `Task<o>` that refer to other objects in the same box as `this`, rather than `this`'s box — to access every required object directly. In both owners-as-dominators and owners-as-modifiers systems, this would typecheck and allow e.g. projects and workers to update their tasks: there is no longer one primary "dominant" decomposition. The problem with this design is that it loses any benefit of ownership types: with everything in one large box, we cannot distinguish between tasks belonging to one project, or another project, or a worker. Once again, a change to one task will be taken as a change to all tasks.

This is where the interpretation of figure 2, modelling boxes as sets, shows us the way out. Just as an element can be a member of *more than one set*, an object can be *inside more than one box*, that is, be owned by more than one object. Where two boxes overlap, objects in their intersection are within both boxes, and so have multiple owners. So all the `Tasks` belonging to $\texttt{Project}_1$, say, will still be inside $\texttt{Project}_1$'s box — $[\![\texttt{Project}_1]\!]$. Similarly, `Tasks` belonging to $\texttt{Worker}_2$ will be inside $[\![\texttt{Worker}_2]\!]$. And, crucially, `Tasks` (or any other object) belonging to both $\texttt{Project}_1$ and $\texttt{Worker}_2$ (for example, $\texttt{Task}_2$ in figure 2) will reside in *both* boxes, that is, in the intersection of the two sets: $[\![\texttt{Project}_1]\!] \cap [\![\texttt{Worker}_2]\!]$. This semantics follows directly from interpreting figure 2 as a set diagram.

The set interpretation generalises equally well to effects in a multiple ownership setting. Read or write effects upon an object with multiple owners must be taken to be effects within the intersection of all the boxes to which that object belongs — and if this intersection itself intersects the effects of another computation, then those two computations potentially interfere.

2.2.1 MOJO: Language Support for Multiple Ownership

We generalise a single-owner language to support multiple owners. Our core language, MOJO, is a relatively simple extension to existing single-ownership languages such as JOE and OGJ [2, 13, 42]; with the simplification that we drop the requirement that owners should be dominators. In the rest of this section we present the various new features of MOJO based on the multiple-ownership version of task management.

First, we reconsider the `Task` class. Surprisingly, this is *exactly the same as the single owner version*. In particular, `Task` retains just one ownership parameter even though in the design — figure 2 — every `Task` has multiple owners. In MOJO, multiple owners are supplied upon class instantiation, rather than upon declaration; therefore classes can be parametric in the number of owners they will have[1].

To instantiate objects with multiple owners, MOJO supports a special ownership combinator that provides multiple (intersection) ownership. The actual ownership argument `a & b` describes multiple owners `a` and `b`: a *single formal* argument is bound by *multiple actual* arguments (similar to a type-generic system, where `List<T>` instantiated by `Pair<A,B>` gives `List<Pair<A,B>>`, and the formal argument `T` is instantiated with a pair of arguments `A` and `B`). For example, we can declare a `Task`

[1] This is an innovation of the current work; in our earlier work [19] multiple class owners were provided upon class declaration.

object that will be owned by a `Worker` and a `Project` object, both previously created:

```
final Project<this> prj = new Project<this>();
final Worker<this>  wrk = new Worker<this>();

Task<prj & wrk> tsk = new Task<prj & wrk>();
```

The interpretation of `a&b` follows clearly from the Objects in Boxes constraint. If an object x's owner is `a&b` then we can assume that there will exist a and b, and x be inside both of them:

$$x \ll a \land x \ll b \Rightarrow x \in [\![a]\!] \cap [\![b]\!]$$

or via the Owners as Boxes constraint:

$$x : \texttt{C<a\&b>} \Rightarrow x \in [\![\texttt{a}]\!] \cap [\![\texttt{b}]\!]$$

The single ownership constraint Single Ownership does *not* hold for multiple ownership. Thus

$$\begin{array}{c} [\![o_1]\!] \cap [\![o_2]\!] \neq \emptyset \\ \not\Rightarrow \\ [\![o_1]\!] \subseteq [\![o_2]\!] \lor [\![o_2]\!] \subseteq [\![o_1]\!] \end{array} \qquad \text{Multiple owners}$$

Returning to our example, the `TaskList` is slightly modified compared to the single owner version

```
class TaskList<o, t0> {
 Task<t0&?> t;
 TaskList<o, t0> next;
 void add(Task<t0 & ?> tt){  ... }
}
```

The owner of each task `t` is now `t0&?`, which says three things: First, that the `Tasks` are inside more than one box — they have multiple ownership. Second, that one of those owners is `t0`, the second owner parameter of the current `TaskList` object. And finally, that — at this point in the program — we do not know what the other owner(s) of each `Task` are.

The code for `Project` class is mostly unchanged,

```
class Project<o> {
 TaskList<this, this> tasks;
 void delay(){
  for(var t : tasks) {t.delay();}
 }
 void add(Task<this & ?> t) {
  tasks.add(t);
 }
}
```

except for the ownership type used to declare the formal parameter of the `add` method.

The `?` wildcard (similar to Java's "?" wildcard for generics) can be thought of as an *existential owner*; these have become common in a range of ownership type systems [18, 31, 50]. Wildcard owners are crucial in a multiple ownership system because one owner often does not, or cannot, know the other potential owners. In our example, the `TaskList` knows that its second owner parameter (`t0`) is one of the owners of the tasks, but does not know who the other owners will be.

The `Worker` class is now symmetrical to the `Project` class:

```
class Worker<o> {
 TaskList<this, this> tasks;
 void delay(){
  for(var t : tasks) {t.delay();}
 }
 void add(Task<this & ?> t) {
  tasks.add(t);
 }
}
```

`Task<p1&w1>` is a subtype of `Task<p1&?>` and of `Task<w1&?>`. This allows us to add tasks owned by, say, project `p1` and worker `w1` to both `p1` and `w1` as in the following code (we discuss the meaning of `intersects` in the following section):

```
final Project<this> p1 = new Project<this>();
final Worker<this>  w1 = new Worker<this>();
w1 intersects p1

Task<p1 & w1> t1 = new  Task<p1 & w1>();
p1.add(t1); w1.add(t1);
```

2.2.2 Effects within Multiple Ownership

Given the straightforward extension from single to multiple ownership promised by the objects in boxes model, it is tempting to expect that effects would generalise similarly; however, that is not quite the case.

In the following example we create two tasks, one shared between project `p1` and worker `w1`, the other shared between `p2` and worker `w1`:

```
class Test {
 final Project<this> p1 = new Project<this>();
 final Project<this> p2 = new Project<this>();
 final Worker<this>  w1 = new Worker<this>();
 w1 intersects p1; w1 intersects p2

 Task<p1 & w1> t1 = new Task<p1 & w1>();
 p1.add(t1); w1.add(t1);

 Task<p2 & w1> t2 = new Task<p2 & w1>();
 p2.add(t2); w1.add(t2); }
```

In this program `p1.delay()` and `w1.delay()` potentially interfere. Given our intuition from the figure 2, we expect `p1.delay()` and `p2.delay()` not to interfere. The expressions have effects:

```
p1.delay()  :  p1 / p1
p2.delay()  :  p2 / p2
w1.delay()  :  w1 / w1
```

and with the machinery we have got so far, we have insufficient information to distinguish the relationship between `p1` and `p2` from that between `p1` and `w1`.

2.2.3 Intersection and Disjointness

To solve this problem we have to provide more information about which boxes intersect, and which boxes are disjoint. Instantiating types with multiple owners like `p1 & w1` creates objects in the set intersection $[\![\texttt{p1}]\!] \cap [\![\texttt{w1}]\!]$, which means that the `p1` box and the `w1` box *must* intersect. Conversely, for disjoint boxes `p1` and `p2` (like in the figure) the multiple owner `p1 & p2` is illegal.

We introduce two declarations that make box topologies explicit. In the example, we'd need to declare `w1 intersects p1` and `w1 intersects p2` if we want to have workers whose tasks are in both `p1` and `p2`. Similarly, we need to declare `p1 disjoint p2` to ensure the `p1` and `p2` boxes are independent. Only one relationship (intersects or disjoint) may be declared between any two boxes: if no relationship is declared, then we don't know what the topology is and we make conservative assumptions.

Then, multiple ownership like `a & b` is legal only if it can be shown that `a` and `b` are legal, and that `a intersects b`. In our example, `p1 & w1` and `w1 & p1` and `p2 & w1` are all legal ("`&`", `intersects` and `disjoint` are symmetric; `intersects` and "`&`" are reflexive; `disjoint` is irreflexive) while `p1 & p2` is **not legal** because `p1` and `p2` are not declared as intersecting.

Effects are independent when we can show that their boxes will be disjoint. For effects involving multiple owners (like `p2 & w1`) it is enough to consider owners pairwise, and to find `one` pair that is definitely disjoint: in the example, `p1` and `p2` are declared to be disjoint, so their intersection is empty, *i.e.* $[\![\texttt{p1}]\!] \cap [\![\texttt{p2}]\!] = \emptyset = [\![\texttt{p1 \& w1}]\!] \cap [\![\texttt{p2 \& w1}]\!] = [\![\texttt{p1 \& ?}]\!] \cap [\![\texttt{p2 \& ?}]\!]$. Therefore `p1.delay()` and `p2.delay()` cannot interfere. On the other hand, because `p1` intersects with `w1`, we are able to create types like `w1 & p1`, while we cannot create `p1 & p2` — the effects $[\![\texttt{w1 \& ?}]\!]$ and $[\![\texttt{p1 \& ?}]\!]$ are **not** independent; thus computations like `w1.delay()` and `p1.delay()` may interfere.

2.2.4 Ownership Type Constraints

To make MOJO modular, we provide `where` clauses to constrain owner parameters. Inside a class `C` with three owner parameters, `a`, `b`, and `o`, we can create objects with ownership `a & o` only if we are sure that `a` intersects with `o`. We give this guarantee through a where clause:

```
class C<o, a, b> where a intersects o {
 Object<a & o> f1; // legal
 Object<a & b> f2; // illegal   }
```

but then we can only instantiate `C` with ownership parameters that are definitively known to intersect. In the example in the previous section, `C<w1,p1,p2>` is legal (because `w1 intersects p2`) while `C<p1,p2,w1>` is illegal because `p1` does not intersect `p2`.

Where clauses can also be used to express disjointness constraints — a declaration such as:

```
class D<o, e> where e disjoint o
{ // ... }
```

requires that the actual ownership parameters be disjoint. In the above example, `D<p1,p2>` is a legal ownership type because `p1 disjoint p2`, but `D<w1,p1>` is not, because those boxes are not disjoint. Note that a disjointness constraint also prevents both parameters being instantiated with the same actual ownership type, because `disjoint` is irreflexive, so `D<p1,p1>` and `D<this,this>` are also illegal.

In practice, we expect that many ownership parameters will use neither intersection nor disjointness constraints. This gives maximal polymorphism: unconstrained parameters can be instantiated with either intersecting or disjoint boxes. A class which does not create objects with multiple owners will not need intersection constraints, and a class which is not susceptible to interference between parameters will not need disjointness constraints. Most collection classes, for example, will fall into this category, as will pairs, tuples, and many other generic classes.

3. MOJO

In this section we present the MOJO language, a minimal object-oriented imperative language, in the Featherweight Java (FJ) [26] style with extensions for (multiple) ownership. It is closely related to JOE [13] and ODE [47].

The major change from FJ is that MOJO types and classes are parameterised by a sequence of owner parameters, the first of which is the owner of objects of that type. Actual ownership parameters may consist of multiple owners which may include the wildcard owner, "?". To support the topology of boxes described in section 2.2.3, constraints on ownership parameters and final fields may be specified.

MOJO supports imperative features, including a heap and field assignment, and final fields that may be used as ownership parameters (non-final fields would be unsafe as ownership parameters as they may change during execution).

The interesting features in MOJO are

- the support for multiple owners, through the operation $\cap$ which combines owners into a "multi-box",
- support for annotations on class declarations, which require disjointness, or allow intersection of ownership parameters,

P	::=	$class^*$	program
$class$	::=	`class` $c < \overline{p} > \ \overline{pCnstr} \ \lhd \ c' < \overline{Q} >$ $\{ \ \overline{finfld} \ \overline{fCnstr} \ \overline{fld} \ \overline{mth} \ \}$	class definition
C	::=	⚭ $\mid$ $\substack{\circ\\\circ}$	interesects or disjoints
$pCnstr$	::=	$p \ C \ p$	parameter constraints
$finfld$	::=	`fin` $t \ \ ff$	final field definition
$fCnstr$	::=	$ff \mathrm{C} ff \mid ff \mathrm{C} p$	field constraints
fld	::=	$t \ \ f$	field definition
mth	::=	$t \ \ m \ (t \ \ x) \ \{ \ \ e \ \ \}$	method body
t	::=	$c < \overline{Q} >$	static type
$path$	::=	`this` $\mid \ x \mid \ \iota \ \mid \ path.ff$	path
Q	::=	$q \ \mid \ q \cap Q$	actual own. param. (poss. multiple)
q	::=	$path \mid$ `?` $\mid \ p$	one actual ownership parameter
R	::=	$r \ \mid \ r \cap R$	runtime actual ownership parameters
r	::=	$\iota \ \mid$ `?`	one runtime actual ownership parameter
e	::=	$x \ \mid$ `this` $\mid \ e.f \ \mid \ \ \mid \ e.f{=}e$ `new` $t \ \mid \ e.m(e) \mid \iota$	expressions
$c, \ p$	::=	id	class identif., form. ownership param.
$f, \ ff, \ m$	::=	id	field identif., final field identif., method identif.

Figure 3. Syntax, runtime entitites in grey.

- support for final fields, and annotations guaranteeing the disjointness and allowing intersection of objects' boxes,
- support for paths appearing as actual ownership parameters in types.

MOJO does not require owners to be dominators, and thus does not provide encapsulation guarantees. The guarantees it provides have to do with the effects of computations.

In comparison to the concrete, surface syntax described in section 2.2.1, the formalism adopts a more succinct abstract syntax: class declarations use $\lhd$ instead of `extends`. Constraints on fields or ownership parameters use ⚭ for `intersects` and $\substack{\circ\\\circ}$ for `disjoint`. To emphasize the connection with set theory, multiple owners use $\cap$ rather than `&`. Actual ownership parameters consist of a set of formal parameters, `this`, final fields, method parameters, the `?` wildcard or, at runtime, addresses. The syntax is given in figure 3.

3.1 Runtime Model

Heaps (h) map addresses to objects. Objects are triples of a runtime type, a mapping from final field identifiers (Id^{ffld}) to addresses, and a mapping from non-final field identifiers (Id^{fld}) to addresses. Runtime types consist of class identifiers and sequences of nonempty sets of addresses, representing actual owners, including `?`[2]

$h \in Heap$	$=$	$\mathbb{N} \longrightarrow Object$	address to object
$Object$	$=$	$c < \overline{R} > \times$	runtime type
		$(\ Id^{ffld} \longrightarrow \mathbb{N}) \ \times$	final fld. values
		$(\ Id^{fld} \longrightarrow \mathbb{N})$	non-final fld. values
$\iota \ \in \mathbb{N}$			object addresses

[2] Allowing `?` gives meaning to the expression `new Task<?,p>`. In MOJO, objects with unknown owners may be desirable, in contrast to Java, where no object is instantiated with wildcard types.

Note, that at runtime, types may mention paths, *e.g.* $c{<}\iota_4.ff_1.ff_2{>}$. We implicitly replace such paths by the lookup of the values of the final fields in the heap, *i.e.* we implicitly apply the following rule whenever required, in order to obtain a $c{<}\overline{R}{>}$ out of a $c{<}\overline{Q}{>}$.

$$\frac{h(\iota)\downarrow_2(ff) = \iota'}{t \ =_h \ [\iota'/\iota.ff]t}$$

3.2 Subclasses, Field and Method Lookup Functions

In figure 4 we define $c{<}\overline{p}{>} \ \ \lhd \ c'{<}\overline{Q'}{>}$, the subclass relation. We can prove that the judgment $c{<}\overline{p}{>} \ \ \lhd \ ...$ implies that the formal parameters of c are $\overline{p}$, and that for given classes c and c', the $c{<}\overline{p}{>} \ \ \lhd c{<}\overline{Q'}{>}$ uniquely determines the $\overline{Q'}$.

Lemma 1.

- *$c \ {<}\overline{p}{>} \ \ \lhd \ c'{<}\_{>}$ implies that* `class` *$c{<}\overline{p}{>}$ ... in the program.*
- *$c \ {<}\overline{p}{>} \ \ \lhd \ c{<}\overline{Q'}{>}$ and $c \ {<}\overline{p}{>} \ \ \lhd \ c'{<}\overline{Q''}{>}$ implies that $\overline{Q''} = \overline{Q'}$.*

Based on the subclass relation, in figure 4 we then define the auxiliary field lookup function $fType^{aux}$ which looks up field types as defined in a class, or as inherited from superclasses. Similarly, we define for the auxiliary method lookup functions $mType^{aux}$ and $mBody^{aux}$.

Subclasses

$$\frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ c'\text{<}\overline{Q'}\text{>}\ \{\ \ldots\ \}}{c\text{<}\overline{p}\text{>}\ \lhd\ c\text{<}\overline{p}\text{>}} \qquad \frac{\texttt{class}\ c\ \text{<}\overline{p}\text{>}\ \ldots\ \lhd\ c'\text{<}\overline{Q'}\text{>}\ \{\ \ldots\ \}}{c\ \text{<}\overline{p}\text{>}\ \lhd\ c'\text{<}\overline{Q'}\text{>}}$$

$$\frac{c\text{<}\overline{p}\text{>}\ \lhd\ c''\text{<}\ \overline{Q''}\ \text{>} \qquad c''\text{<}\overline{p''}\text{>}\ \lhd\ c'\text{<}\ \overline{Q'}\ \text{>}}{c\text{<}\overline{p}\text{>}\ \lhd\ c'\text{<}\ \overline{[\overline{Q''/p''}]Q'}\ \text{>}}$$

Field lookup

$$\frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ \ldots\ \{\ \ldots tf \ldots\ \}}{fType^{aux}(c\text{<}\overline{p}\text{>}, f) = t} \qquad \frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ \ldots\ c'\text{<}\ \overline{Q'}\ \text{>} \quad fType^{aux}(c'\text{<}\overline{p'}\text{>}, f) = t}{fType^{aux}(c\text{<}\overline{p}\text{>}, f) = [\overline{Q'/p'}]t}$$

$$allFields(c) = \{\ f\ \mid\ fType^{aux}(c\text{<}\overline{p}\text{>}, f) \text{ is defined for some } \overline{p}\}$$
$$finFields(c) = allFields(c)\ \cap\ \{\ f\!f \mid f\!f \text{ is final}\}$$
$$nonfinFields(c) = allFields(c)\ \cap\ \{\ f \mid f \text{ is not final}\}$$

$$fType(c\text{<}\overline{Q}\text{>}, f, e, \Gamma) = [\overline{Q/p}](t^{\Gamma\cdot e}) \qquad \text{where } t = fType^{aux}(c\text{<}\overline{p}\text{>}, f)$$
$$t^{\Gamma\cdot e} = \begin{cases} t, & \text{if } \texttt{this} \notin t; \\ [path/\texttt{this}]t & \text{if } \texttt{this} \in t, e \text{ is a } path \text{ in } \Gamma, \\ \bot, & \text{otherwise.} \end{cases}$$

Method lookup

$$\frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ \ldots\ \{\ \ldots t\ m(t'\ x)\{\ldots\}\ldots\ \}}{mType^{aux}(c\text{<}\overline{p}\text{>}, m) = t'\ \to t} \qquad \frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ \ldots\ c'\text{<}\ \overline{Q'}\ \text{>} \quad mType^{aux}(c'\text{<}\overline{p'}\text{>}, m) = t' \to t}{mType^{aux}(c\text{<}\overline{p}\text{>}, m) = [\overline{Q'/p'}]t'\text{−>}\ [\overline{Q'/p'}]t}$$

$$\frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ \ldots\ \{\ \ldots t\ m(t'\ x)\{\ e\ \}\ldots\ \}}{mBody^{aux}(c\text{<}\overline{p}\text{>}, m) = (x, e)} \qquad \frac{\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots\ \lhd\ c'\text{<}\ \overline{Q'}\ \text{>}\ldots}{mBody^{aux}(c\text{<}\overline{p}\text{>}, m) = [\overline{Q'/p'}]mBody^{aux}(c'\text{<}\overline{p'}\text{>}, m)}$$

$$mBody(c\text{<}\overline{Q}\text{>}, m) = [\overline{Q/p}]mBody^{aux}(c\text{<}\overline{p}\text{>}, m)$$

Figure 4. Subclasses, field, and method lookup functions.

$$\frac{}{v, h\ \rightsquigarrow\ v, h} \qquad \frac{e, h\ \rightsquigarrow\ \iota, h'}{e.f\!f, h\ \rightsquigarrow\ h'(\iota)\downarrow_2 (f\!f), h'} \qquad \frac{e, h\ \rightsquigarrow\ \iota, h'}{e.f, h\ \rightsquigarrow\ h'(\iota)\downarrow_3 (f), h'}$$

$$\frac{e, h\ \rightsquigarrow\ \iota, h'' \qquad e', h''\ \rightsquigarrow\ \iota', h'}{e.f = e', h\ \rightsquigarrow\ \iota', h'[\iota \mapsto (h'(\iota)\downarrow_1, h'(\iota)\downarrow_2, h'(\iota)\downarrow_3 [f \mapsto \iota']\ )\ ]}$$

Figure 5. Operational semantics for field access and field assignment

Lemma 2.

- *$c\text{<}\overline{p}\text{>}\ \lhd\ c'\ \text{<}\overline{Q'}\text{>}$, and $fType^{aux}(c'\text{<}\overline{p'}\text{>}, f) = t$ implies that $fType^{aux}(c\text{<}\overline{p}\text{>}, f) = [\overline{Q'/p'}]t$*
- *$c\text{<}\overline{p}\text{>}\ \lhd\ c'\text{<}\ \overline{Q'}\ \text{>}$, implies $mType^{aux}(c'\text{<}\overline{p'}\text{>}, f) = t' \to t$, then $mType^{aux}(c\text{<}\overline{p}\text{>}, m) = [\overline{Q'/p'}]t' \to [\overline{Q'/p'}]t$.*
- *If $fType^{aux}(t, \_)$ or $mType^{aux}(t, \_)$, or $mBody^{aux}(t, \_)$ is defined, then $t = c\text{<}\overline{p}\text{>}$ and $\texttt{class}\ c\text{<}\overline{p}\text{>}\ \ldots$ in the program, for some c and $\overline{p}$.*

The functions $allFields$, $finFields$ and $nonfinFields$ return, respectively, the identifiers of all the fields of a class, all final fields of a class, all non-final fields defined in a class.

The function $fType(c{<}\overline{Q}{>}, f, e, \Gamma)$ returns the type of field f as accessed from e, which has type $c{<}\overline{Q}{>}$, in an environment Γ. It first obtains the type of the field as defined in class c (using the function $fType^{aux}$, then it replaces any occurrences of `this`, provided that e is a path (using the operation $t^{\Gamma \cdot e}$), and then replaces the formal owner parameters $\overline{p}$ by the actual owner parameters Q. For example, $fType(\texttt{Worker<o>}, \texttt{tasks}, \texttt{w}, \Gamma)$ is `TaskList<w,w>`[3].

3.3 Execution

Execution is defined in terms of a large steps operational semantics, with format $e, h \leadsto v, h'$, which maps an expression and a heap to a result and a new heap.

The operational semantics for field assignment and field access is the obvious one and appears in figure 5. The semantics of object creation and method call is more intricate, and we discuss it here in more detail.

To create an object of type $c < \overline{R} >$, we first create a new object at a fresh address ι with temporary type $Object$[4]. We then initialize the final fields $\mathit{ff}_1, ... \mathit{ff}_n$ of c and obtain objects $\bar{\iota}$, and a heap h_n. We then initialize the non-final fields f_1,...f_m and obtain objects $\overline{\iota'}$, and a heap h'_m. Finally, in h'_m we update the class of the new object, and "connect" the final field identifiers to $\bar{\iota}$, and the non-final field identifiers to $\overline{\iota'}$.

$$\frac{\begin{array}{c} \iota \text{ fresh in } h \qquad h_1 = h[\iota \mapsto (Object, \emptyset, \emptyset)] \\ finFields(c{<}\overline{R}{>}) = \mathit{ff}_1, ...\mathit{ff}_n \\ \texttt{new}\ fType(c{<}\overline{R}{>}, \mathit{ff}_i, \iota, h), h_i \ \leadsto\ \iota_i, h_{i+1} \qquad i \in 1,..n \\ h'_1 = h_{n+1} \qquad nonfinFields(c{<}\overline{R}{>}) = f_1, ...f_m \\ \texttt{new}\ fType(c{<}\overline{R}{>}, f_i, \iota, h), h'_i \ \leadsto\ \iota'_i, h'_{i+1} \qquad i \in 1,..m \end{array}}{\texttt{new}\ c < \overline{R} >, h \ \leadsto\ \iota, h'_m[\iota \mapsto (c < \overline{R} >, \overline{\mathit{ff} \mapsto \iota}, \overline{f \mapsto \iota'})]}$$

Method calls evaluate the receiver and the argument, and look up the method body in the class as usual. More interestingly, in e_3, the method body, we replace the formal receiver by the actual one ($\iota/\texttt{this}$), and the formal parameter by the actual one (ι'/x). The class's ownership parameters will have already been replaced by the corresponding sets of owners in the object's runtime type $(\overline{R/p})$ by the $mBody$ function.

[3] The order of the last two operations in the definition of $fType$ is crucial; if the order was reversed, then the $[\overline{Q/p}]$ could introduce `this` into the type, which would be incorrectly replaced (free variable capture) by the $t^{\Gamma \cdot e}$ operation. For example, in:

```
class A<a>{ B<a> f; }
class C<c>{ final A<this> a1; ... a1.b ... }
```

the type of `a1.b` is `B<this>`. However, reversing the two operations would give to `a1.b` the (wrong) type `B<a1>`.

[4] We do not give the newly created object the class $c{<}\overline{R}{>}$, in order to avoid objects with uninitialized final fields. We give ι the type $c{<}\overline{R}{>}$ only after the values for all new fields are available.

$$\frac{\begin{array}{c} e_1, h \leadsto \iota, h'' \qquad e_2, h'' \leadsto \iota'', h''' \\ h'''(\iota) = (c < \overline{R} >, ..., ...) \\ mBody(m, c < \overline{R} >) = (x, e_3) \\ [\iota/\texttt{this}, \iota''/x]e_3, h''' \leadsto \iota', h' \end{array}}{e_1.m(e_2), h \leadsto \iota', h'}$$

3.4 Well-formed types

In figure 6 we define the following five judgments:

$\Gamma \vdash q \ll q'$	q guaranteed to be inside q'
$\Gamma \vdash Q \mathrel{\circ\!\!\circ} Q'$	Q allowed to intersect Q'
$\Gamma \vdash Q \mathrel{\genfrac{}{}{0pt}{}{\circ}{\circ}} Q'$	Q guaranteed disjoint with Q'
$\Gamma \vdash Q$	Q consists of qs allowed to intersect
$\Gamma \vdash c < \overline{Q} >$	$c < \overline{Q} >$ well-formed type

An environment, Γ, maps `this`, x and ι to types, and contains a set of formal ownership parameters (p) and intersects and disjoints relationships declared in the class of the receiver.

The operator $\cap$ is associative and commutative, and the empty sequence ϵ is neutral, *i.e.* $\epsilon \cap Q = Q$.

An object is inside another, if its box (that is, the set of objects it owns) is a subset of the box of the other.

The relations $\circ\!\!\circ$ and $\genfrac{}{}{0pt}{}{\circ}{\circ}$ extend the declared intersections and disjointness of owner parameters and fields. The `disjoint` relation makes use of the inside ($\ll$) relation for owner parameters.

A type $c < \overline{Q} >$ is well-formed in the context of an environment Γ, iff: a) there is a Q for each formal parameter p; b) each Q is well-formed (*i.e.* consists of ownership parameters which are allowed to intersect); and c) if two parameters are declared to intersect or be disjoint in the class declaration, then the environment Γ will allow the parameters to intersect or guarantee them to be disjoint, respecively.

3.5 Subtypes

In figure 7 we define the subtype relation $t' <: t$ which is based on the subclass relationship. The auxiliary judgment $Q \sqsubseteq Q'$ guarantees that Q' is the same as Q except that some of the contents of Q may be replaced by ?. Note that $\_ \sqsubseteq \_$ is reflexive and transitive, but *not* symmetric. We can easily prove that subtyping is transitive.

For types $c{<}\overline{Q}{>}$ and $c{<}\overline{Q'}{>}$, if no `?` appears in $\overline{Q}$ or $\overline{Q'}$, subtyping is invariant with respect to the ownership parameters. For example, $\texttt{C} <\texttt{o}_1 \cap \texttt{o}_2>$ is not a subtype of $\texttt{C} <\texttt{o}_1>$ — to allow such a relation would be unsound. Similarly to Java Wildcards [11, 49], the `?` owner introduces variance (with respect to ownership parameters, as opposed to type parameters). However, in MOJO, `?` also denotes variance in the *number of owners*. For example, as well as the obvious relationship $\texttt{C} <\texttt{o}> <: \texttt{C} <?>$, we also have $\texttt{C} <\texttt{o}_1 \cap \texttt{o}_2> <: \texttt{C} <?>$[5].

[5] $Q \cap ? \sqsubseteq Q$ is not part of the subtyping rules, it is not sound because it would allow us to 'add variance' to an invariant type.

Objects allowed to intersect, or guaranteed to be disjoint

$$\frac{}{\Gamma \vdash q \mathbin{\circ\!\circ} q} \qquad \frac{\Gamma \vdash q' \; C \; q}{\Gamma \vdash q \; C \; q'} \qquad \frac{}{\Gamma \vdash q \mathbin{\circ\!\circ} ?} \qquad \frac{q \; C \; q' \in \Gamma}{\Gamma \vdash q \; C \; q'} \qquad \frac{\Gamma \vdash q : t \quad f\!f \; C \; f\!f' \in fCnstrs(t)}{\Gamma \vdash q.f\!f \; C \; q.f\!f'}$$

$$\frac{\Gamma \vdash q \ll q'}{\Gamma \vdash q \mathbin{\circ\!\circ} q'} \qquad \frac{\Gamma \vdash q' \ll q'' \quad \Gamma \vdash q'' \mathbin{\overset{\circ}{\circ}} q}{\Gamma \vdash q \mathbin{\overset{\circ}{\circ}} q'}$$

$$\frac{Q = Q_1 \cap q \quad Q' = Q_2 \cap q' \quad \Gamma \vdash q \mathbin{\overset{\circ}{\circ}} q'}{\Gamma \vdash Q \mathbin{\overset{\circ}{\circ}} Q'} \qquad \frac{Q = Q_1 \cap q, \; Q' = Q_2 \cap q' \implies \Gamma \vdash q \mathbin{\circ\!\circ} q'}{\Gamma \vdash Q \mathbin{\circ\!\circ} Q'}$$

Well-formed types

$$\frac{q \in \mathcal{D}m(\Gamma)}{\Gamma \vdash q} \qquad \frac{}{\Gamma \vdash ?} \qquad \frac{\Gamma \vdash q : t \quad f\!f \in fFields(t)}{\Gamma \vdash q.f\!f} \qquad \frac{\Gamma \vdash Q \quad \Gamma \vdash q \quad Q = Q' \cap q' \implies \Gamma \vdash q \mathbin{\circ\!\circ} q'}{\Gamma \vdash Q \cap q}$$

$$\frac{\texttt{class}\ c < \overline{p} > \ \ldots \ \triangleleft \ldots \qquad |\overline{Q}| = |\overline{p}| \qquad \Gamma \vdash \overline{Q} \qquad Q_i \; C \; Q_j \in pCnstrs(c < \overline{Q} >) \implies \Gamma \vdash Q_i \; C \; Q_j}{\Gamma \vdash c < \overline{Q} >}$$

Inside relation for owner parameters

$$\frac{}{\Gamma \vdash q \ll q} \qquad \frac{\Gamma \vdash q \ll q'' \quad \Gamma \vdash q'' \ll q'}{\Gamma \vdash q \ll q'} \qquad \frac{\Gamma(q) = c < q' \cap Q, \overline{Q'} >}{\Gamma \vdash q \ll q'}$$

Figure 6. Well-formed types and the 'inside', intersects and disjoint relations for owner parameters

Wildcards in both MOJO and Java are a *use-site* variance mechanism, as opposed to *declaration-site* variance, found in, for example, Scala [39] (again in the contex of type, not ownership, parameters).

3.6 Types of expressions

The type of an expression e depends on an environment Γ and is given by the judgment $\Gamma \vdash e : t$ defined in figure 7. The rules are as expected for an ownership type system, with some special care taken for field assignment and parameter passing when the types involve ?, this is done using the 'strict' versions of the field and method type functions, also given in figure 7. Consider the following classes:

```
class B<b1>{  ...  }
class C<c1>{  B<c1> f1; B<?> f2; }
```

in the example:

```
class Test<t1,t2>{
  void m1(C<t1> x, C<?> y){
    x.f2 = new B<t2>;   // type correct
    x.f2 = new B<?>;    // type correct
    y.f1 = new B<t2>;   // type error
    y.f1 = new B<?>;    // type error
    y.f2 = new B<?>;    // type correct
  }
}
```

the assignments to `x.f2` are type correct because from the point of view of `x` its field `f2` may contain a `D<`Q`>`, for *any* actual owners Q. On the other hand, any assignment to `y.f1` is type-incorrect, because from the point of view of `y` its field `f1` must contain a `D<`Q`>`, for some *fixed* actual owners Q, which are unknown in the current context. In terms of our formal description, the first two and the last assignment are type correct, because for all Q, it holds that $[Q/\texttt{c1}]^{strict}$`B<any>` = `B<any>`; the next two assignments are type incorrect, because $[?/\texttt{c1}]^{strict}$`B<c1>` is undefined.

Furthermore, the types of fields and methods need to treat the actual ownership parameter `this` specially; ie it replaces `this` by the expression whose field or method is being selected, provided that e denotes a constant value. This is described through $t^{\Gamma \cdot e}$, defined in figure 4, where `this` $\in t$ means that `this` appears in t:

The following lemma is used to prove soundness of the type system (in Theorem 1 for the cases of field assignment and method call):

Lemma 3. *If e and e' are paths evaluating to the same the address in heap h, and $[\overline{Q''/p}]^{strict}t \neq \bot$, and $[\overline{Q'/p'}]Q_i \leq Q''_i$ for all i, then:*
$[\overline{Q'/p'}]((([\overline{Q/p}]t)^{h \cdot e}) \;=_h\; [\overline{Q''/p}](t^{h \cdot e'})$.

Note that without the requirement $[\overline{Q''/p}]^{strict}t \neq \bot$, the left hand side type would have been a pure subtype of the righthand side.

Subtypes

$$\frac{}{q \sqsubseteq q} \qquad \frac{}{q \sqsubseteq ?} \qquad \frac{Q \sqsubseteq Q' \qquad q \sqsubseteq q'}{Q \cap q \sqsubseteq Q' \cap q'} \qquad \frac{}{Q \sqsubseteq Q \cap ?}$$

$$\frac{c\langle\overline{p}\rangle \lhd c'\langle\overline{Q'}\rangle}{c\langle\overline{Q}\rangle <: c'\langle[\overline{Q/p}]\overline{Q'}\rangle} \qquad \frac{\overline{Q \sqsubseteq Q'}}{c\langle\overline{Q}\rangle <: c\langle\overline{Q'}\rangle}$$

Strict method and field type lookup

$$[\overline{Q/p}]^{strict}t \;=\; \begin{cases} [\overline{Q/p}]t, & \text{if } \; ?{\in}Q_i \Rightarrow p_i \notin t \\ \bot, & \text{otherwise.} \end{cases}$$

$$\mathit{fType}^{strict}(c\langle\overline{Q}\rangle, f, e, \Gamma) = [\overline{Q/p}]^{strict}t', \qquad \text{where} \quad t = \mathit{fType}^{aux}(c\langle\overline{P}\rangle, f) \;\text{ and }\; t' = t^{\Gamma \cdot e}.$$

$$\mathit{mType}^{strict}(c\langle\overline{Q}\rangle, m, e, \Gamma) = [\overline{Q/p}]^{strict}t3 \rightarrow [\overline{Q/p}]t_4,$$
$$\text{where } t_1 \rightarrow t_2 = \mathit{mType}^{aux}(c\langle\overline{p}\rangle, m) \;\text{ and }\; t_3 = {t_1}^{\Gamma \cdot e} \;\text{ and }\; t_4 = {t_2}^{\Gamma \cdot e}.$$

Types of Expressions

$$\frac{}{\Gamma \vdash q : \Gamma(q)} \qquad \frac{\Gamma \vdash t}{\Gamma \vdash \texttt{new}\ t : t} \qquad \frac{\Gamma \vdash e : t' \qquad t' <: t}{\Gamma \vdash e : t}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \\ \mathit{fType}(t, f, e, \Gamma) = t'\end{array}}{\Gamma \vdash e.f : t'} \qquad \frac{\begin{array}{c}\Gamma \vdash e : t \\ \mathit{fType}^{strict}(t, f, e, \Gamma) = t' \\ \Gamma \vdash e' : t'\end{array}}{\Gamma \vdash e.f{=}e' : t'} \qquad \frac{\begin{array}{c}\Gamma \vdash e : t \\ \mathit{mType}^{strict}(t, m, e, \Gamma) = t' \rightarrow t'' \\ \Gamma \vdash e' : t'\end{array}}{\Gamma \vdash e.m(e') : t''}$$

Figure 7. Subtypes, and Typing rules.

3.7 Well-formed Class and Program

A class is well-formed if it has the same owner as the superclass, the superclass type is well-formed, types mentioned in fields and methods are well-formed, and method bodies are well typed. For checking well-formedness of the superclass, constraints between ownership parameters are taken into account. For checking types of final fields, `this` is allowed to appear in $\overline{t}$. For checking types of fields and method bodies, constraints between final fields are also taken into account; this happens implicitly, through the introduction of `this` $c\,\langle\overline{p}\rangle$ into the environment Γ''. Fields must not overlap with those from the superclass. Finally, the constraints on ownership parameters and final fields must be well-formed.

$$\frac{\begin{array}{c}\texttt{class}\ c\langle\overline{p}\rangle\ \overline{pCnstr}\ \lhd\ c'\langle\overline{Q}\rangle\ \{\ \overline{\texttt{fin}\ t\ \mathit{ff}}\ \overline{fCnstr}\ \overline{t'\ f}\ \overline{mth}\ \} \\ Q_1 = p_1 \\ \Gamma = \overline{p}, pCnstrs(c\,\langle\overline{p}\rangle) \qquad \Gamma \vdash c' < \overline{Q} > \\ \Gamma' = \Gamma, \texttt{this} \qquad \Gamma' \vdash \overline{t} \\ \Gamma'' = \Gamma', \texttt{this}\ c\,\langle\overline{p}\rangle \qquad \Gamma'' \vdash \overline{t'} \qquad \Gamma'' \vdash \overline{mth} \\ finFields(c'\langle\overline{Q}\rangle) = \overline{t''\ \mathit{ff}'} \qquad \overline{\mathit{ff}'} \cap \overline{\mathit{ff}} = \emptyset \\ nonfinFields(c'\langle\overline{Q}\rangle) = \overline{t'''\ f'} \qquad \overline{f'} \cap \overline{f} = \emptyset \\ \vdash \Gamma''\diamond \end{array}}{c < \overline{p} > \;\text{ well formed}}$$

Because MOJO does not require owners to be dominators, there is no need for annotations requiring that certain owner parameters should be inside others. Thus, we have omitted them for simplicity, although such annotations allow more information about disjointness to be deduced, and should be part of a full language.

The constraints on owner parameters and fields are well-formed if they contain no contradictions:

$$\frac{\begin{array}{c}\Gamma \vdash p \mathrel{\circ\!\!\circ} p' \Rightarrow \Gamma \not\vdash p \overset{\circ}{\circ} p' \\ \Gamma \vdash p \overset{\circ}{\circ} p' \Rightarrow \Gamma \not\vdash p \mathrel{\circ\!\!\circ} p'\end{array}}{\vdash \Gamma\diamond}$$

A method body is well typed if it contains an expression of the same type as the return type of the method, and if overriding is legal (defined in figure 9).

$$\frac{\begin{array}{c}\Gamma(this) = c\,\langle\overline{p}\rangle \\ \texttt{class}\ c\,\langle\overline{p}\rangle\,\overline{pCnstr}\ \lhd\ c' < \overline{Q} > ... \\ \Gamma \vdash t \qquad \Gamma \vdash t' \qquad \Gamma, t'\ x \vdash e : t \\ override(m, c' < \overline{Q} >, t' \rightarrow t)\end{array}}{\Gamma \vdash t\ m(t'\ x)\{e\}}$$

3.8 Runtime Types

The function *env* maps heaps to typing environments, enriching these with information about the values of final fields:

Definition 1. *We define env as follows:*

- $env(\overline{\iota \mapsto obj}) = \overline{env(\iota \mapsto obj)}$
- $env(\iota \mapsto obj) =$
  $\{\ \iota : c <\!\overline{R}\!>\ \} \ \cup\ \{\ \iota.\overline{f\!f} \mapsto \overline{\iota}\ \} \ \cup$
  $\{\ (\iota_i\ C\ \iota_j)\ \ |\ f\!f_i\ C\ f\!f_j \in fCnstrs(c <\!\overline{R}\!>)\}$
  where $obj = (c <\!\overline{R}\!>, \overline{f\!f \mapsto \iota}, \ldots)$

Wherever an environment gives a judgment, a corresponding heap gives the same judgment:

$$h \vdash judg\_x \iff env(h) \vdash judg\_x$$

Thus we obtain judgements for typing expressions, well-formed types, the inside relation, etc:

$$h \vdash \iota \ll \iota' \qquad h \vdash Q \infty Q' \qquad h \vdash Q \mathbin{\substack{\circ\\\circ}} Q' \qquad \vdash h\diamond$$
$$h \vdash Q \qquad h \vdash c <\!Q\!> \qquad h \vdash e : t$$

3.9 Well-formed Heap

$$[\![\ \iota\ ]\!]_h = \{\iota' \mid h \vdash \iota' \ll \iota\ \}$$

$$\frac{\begin{array}{c} h(\iota) = (c<\!\overline{R}\!>, \_, \_) \qquad h \vdash c<\!\overline{R}\!> \\ fType(c<\!\overline{R}\!>, f, \iota, h) = t \implies h \vdash h(\iota)(f) : t \end{array}}{h \vdash \iota}$$

$$\frac{\begin{array}{c} \forall \iota \in \mathcal{D}m(h):\ h \vdash \iota \\ [\![\ \iota\ ]\!]_h \cap [\![\ \iota'\ ]\!]_h \neq \emptyset \implies h \vdash \iota \infty \iota' \\ h \vdash \iota \mathbin{\substack{\circ\\\circ}} \iota' \implies [\![\ \iota\ ]\!]_h \cap [\![\ \iota'\ ]\!]_h = \emptyset \\ \vdash h \diamond \end{array}}{\vdash h}$$

Figure 8. Well-formed objects and heaps

In figure 8 we define well-formed objects and heaps. An object ι in the heap is well-formed, expressed by $h \vdash \iota$, if its type is well-formed, and all its fields have types according to their static types. The heap is well-formed if all objects in the heap are well-formed; where the boxes of objects intersect in the heap, the heap can show that these objects are in a ∞ relationship; if the heap can show that two objects are in a $\substack{\circ\\\circ}$ relationship, then their boxes do not overlap; finally, the heap must contain no contradictions, *i.e.* there exist no objects ι, ι' such that $h \vdash \iota \infty \iota'$ and $h \vdash \iota \mathbin{\substack{\circ\\\circ}} \iota'$.

3.10 Soundness of the Type System

Theorem 1. *For a well formed program, if* $h \vdash e : t$ *and* $\vdash h$ *and* $e, h \rightsquigarrow \iota, h'$*, then* $h' \vdash \iota : t$*, and* $\vdash h'$.

Proof. By structural induction, and using lemmas 6, 7, 9, 10 and 11, listed in this section.

The proofs of lemmas 6, 7, 9, 10, and 11 use further auxiliary lemmas.

Lemma 4 (Inversion Lemma)**.** *We define the inversion lemma in the usual way, that is, in a well-formed program, if* $\Gamma \vdash e : t$ *then the premises of the appropriate type rule holds. The only case that is interesting is where* $e = q$*, in which case we must insist that* $\Gamma(q)$ *is defined. This ensures* q *is not a path, but a variable.*

Lemma 5. *In a well-formed program, if* $\vdash h$ *and* $h \vdash e : t$ *then*

1. *neither* x *nor* `this` *appear in* e*;*
2. $t = c <\!\overline{R}\!>$.

We first prove that runtime types and the inside relation are invariant with execution, while disjointness and possible intersection of objects are monotonic:

Lemma 6. *In a well-formed program, if* $\iota, \iota' \in \mathcal{D}m(h)$*, and* $e, h \rightsquigarrow \iota'', h'$*, and* $h \vdash e : t'$*, then*

1. $h \vdash \iota\ :\ t$ *if and only if* $h' \vdash \iota\ :\ t$
2. $h \vdash \iota\ \ll \iota'$ *if and only if* $h' \vdash \iota\ \ll \iota'$
3. $h \vdash \iota\ C\ \iota'$ *if and only if* $h' \vdash \iota\ C\ \iota'$

As usual in soundness proofs, we need a substitution lemma; in our particular setting, the substitution needs to be aware of ownership and allowed/forbidden intersections.

For a substitution σ which maps `this` and x to addresses, we define its expansion, σ_h, so that it also maps formal ownership parameters ($\overline{p}$). We then define the concept of an appropriate substitution $\Gamma, h \vdash \sigma$, as one which preserves all constraints impled in Γ:

Definition 2. *Given a* $\sigma : \{\texttt{this}, x\} \longrightarrow \mathbb{N}$*, we define:*

- $\sigma_h\ :\ id \longrightarrow \mathcal{P}wr(\mathbb{N})$ *as follows:*
  1. $\sigma_h(\texttt{this}) = \sigma(\texttt{this}),\ \ \sigma_h(x) = \sigma(x)$.
  2. $\sigma_h(p) = R_i$ *if* $h(\sigma(\texttt{this})) = (c < \overline{R} >, \ldots, \ldots)$ *and* $\mathcal{D}m(c) = \overline{p}$ *and* $p = p_i$*; undefined otherwise.*
- $\sigma_h \circ t$ *indicates the application of* σ_h *on type* t.
- $\sigma_h \circ e$ *indicates application of* σ_h *on expression* e.
- $\Gamma, h \vdash \sigma$ *iff for any constraint* C*:*
  1. $q\ C\ q' \in \Gamma \implies h \vdash \sigma_h(q)\ C\ \sigma_h(q')$,
  2. $h \vdash \sigma(\texttt{this}) : \sigma_h \circ \Gamma(\texttt{this})$,
  3. $h \vdash \sigma(x) : \sigma_h \circ \Gamma(x)$,
  4. $p \in \mathcal{D}m(\Gamma) \implies \sigma_h(p) \subseteq \mathcal{D}m(h)$,
  5. $p \in \mathcal{D}m(\Gamma),\ \iota, \iota' \in \sigma_h(p) \implies h \vdash \iota \infty \iota'$.

We can now prove the substitution lemma:

Lemma 7. *If* $\Gamma, h \vdash \sigma$ *then:*

1. $\Gamma \vdash q \ll q'$ *implies* $h \vdash \sigma_h(q) \ll \sigma_h(q')$.
2. $\Gamma \vdash q\ C\ q'$ *implies* $h \vdash \sigma_h(q)\ C\ \sigma_h(q')$.
3. $\Gamma \vdash Q\ C\ Q'$ *implies* $h \vdash \sigma_h(Q)\ C\ \sigma_h(Q')$.

Constraint lookup and method override

$$\frac{}{pCnstrs(\texttt{Object}) = \emptyset} \qquad \frac{\texttt{class}\ c\ <\overline{p}>\ \overline{pCnstr}\ \lhd\ c'<\overline{Q'}>\ \ldots}{pCnstrs(c<\overline{Q}>) = [\overline{Q/p}]\overline{pCnstr}, pCnstrs([\overline{Q/p}]c'<\overline{Q'}>)}$$

$$\frac{}{fCnstrs(\texttt{Object}) = \emptyset} \qquad \frac{\texttt{class}\ c<\overline{p}>\ \overline{pCnstr}\ \lhd\ c'<\overline{Q'}>\ \{\ \overline{\texttt{fin}\ t\ ff}\ \overline{fCnstr}\ \overline{t'\ f}\ \overline{mth}\ \}}{fCnstrs(c<\overline{Q}>) = [\overline{Q/p}]\overline{fCnstr}, fCnstrs([\overline{Q/p}]c'<\overline{Q'}>)}$$

$$\frac{mType(m, c<\overline{Q}>)\ undefined}{override(m, c<\overline{Q}>, t' \to t)} \qquad \frac{mType(m, c<\overline{Q}>) = t' \to t}{override(m, c<\overline{Q}>, t' \to t)}$$

Figure 9. Constraint lookup functions and overrride function.

4. $t' <: t$ *implies* $\sigma_h(t') <: \sigma_h(t)$.
5. $\Gamma \vdash q : t$ *implies* $h \vdash \sigma_h(q) : \sigma_h(t)$.
6. $\Gamma \vdash t$ *implies* $h \vdash \sigma_h(t)$.
7. $\Gamma \vdash e : t$ *implies* $h \vdash \sigma_h(e) : \sigma_h(t)$.

We define $t^{[c \lhd c']}$ the "projection" of a type t as seen from a class c to the way it is seen from a subclass c', and similarly, of an environment or an expression:

Definition 3. *For environment* Γ, *classes* c, c', *type* t, *expression* e, $\overline{p} = \mathcal{D}m(c)$, $\overline{p'} = \mathcal{D}m(c')$, *we define:*

$$\begin{array}{ll}
e^{[c \lhd c']} = & [\overline{Q/p}]e, \textit{ if } c' <\overline{p'}> <:\ c <\overline{Q}> \\
& \textit{undefined}, \quad \textit{otherwise}. \\
t^{[c \lhd c']} = & [\overline{Q/p}]t, \textit{ if } c' <\overline{p'}> <:\ c <\overline{Q}> \\
& \textit{undefined}, \quad \textit{otherwise}. \\
\Gamma^{[c \lhd c']} = & [\overline{Q/p}]\overline{t'\ x}, c' <\overline{p'}>\ \texttt{this}, \overline{p'}, \overline{pConstr'} \\
& \textit{if } c' <\overline{p'}> <:\ c <\overline{Q}> \textit{ for some } \overline{Q}, \textit{ and} \\
& \Gamma = \overline{t'\ x}, c <\overline{p}>\ \texttt{this}, \overline{p}, \overline{pConstr}, \\
& \textit{and } \overline{pConstr'} = [\overline{Q/p}]\overline{pConstr}. \\
& \textit{undefined}, \quad \textit{otherwise}.
\end{array}$$

We then prove that projection to subclasses preserves typing. In other words, if a type t is well formed in an environment from class c, then the projection of t onto the subclass c' is well-formed in the environment as defined in the subclass c'.

Lemma 8. *For classes* c, *and* c', *environments* Γ *so that* $\Gamma^{[c \lhd c']}$ *is defined:*

- $t <: t'$ *implies* $t^{[c \lhd c']} <: t'^{[c \lhd c']}$.
- $\Gamma \vdash q : t$ *implies* $\Gamma^{[c \lhd c']} \vdash q^{[c \lhd c']} : t^{[c \lhd c']}$.
- $\Gamma \vdash t$ *implies* $\Gamma^{[c \lhd c']} \vdash t^{[c \lhd c']}$.
- $\Gamma \vdash e : t$ *implies* $\Gamma^{[c \lhd c']} \vdash e^{[c \lhd c']} : t^{[c \lhd c']}$.

Using the above lemma we can prove that in well-formed programs the method body always satisfies its type:

Lemma 9. *If* $mType^{aux}(c < \overline{p} >, m) = t_1 \to t_2$ *and* $mBody^{aux}(c<\overline{p}>, m) = (x, e)$, *and* $\Gamma = \texttt{this} : c<\overline{p}>, x : t_1, \overline{p}, pCnstrs(c<\overline{p}>)$, *then* $\Gamma \vdash e : t_2$.

We prove that subclassing preserves method types:

Lemma 10. *In a well-formed program, if* $c < \overline{p} > \lhd c' < \overline{Q'} >$ *and* $mType^{aux}(c' < \overline{p'} >, m) = t_1 \to t_2$ *and* `class` $c' < \overline{p'} >\ \ldots$ *then* $mType^{aux}(c < \overline{p} >, m) = [\overline{Q'/p'}]t_1 \to [\overline{Q'/p'}]t_2$.

Lemma 11. *In a well-formed program, if* $e, h \rightsquigarrow \iota', h'$ *and* $h(\iota) = (t, \ldots)$ *then* $h'(\iota) = (t, \ldots)$.

4. Effects

Effects are used to give a conservative estimate of the area of the heap read or written by an expression. We describe these areas through one or more boxes, where the "$\cup$" operator describes the union of such boxes. The first part of an effect is the area being read; the second is the area being written:

$$\begin{array}{lll}
\phi & ::= \ \epsilon \ \mid \ Q \cup \phi & \textit{boxes} \\
\psi \in \textit{Effect} & ::= \ \phi/\phi & \textit{effect}
\end{array}$$

We expect programs to come equipped with a function to give us the effects of a method[6]:

$$\mathcal{M}_{\mathit{eff}}(\_,\_) : Id^{class} \times Id^{mth} \longrightarrow \textit{Effect}$$

4.1 The Example with Effects

We now revisit the example from section 2, and give the values for the function $\mathcal{M}_{\mathit{eff}}(\_,\_)$ through comments in the code.

```
class Duration<d1> {
    Date<this> start;  Date<this> end;
    void delay(){...}     // EFF: this / this
}

class Task<t1> {
    Duration<this> duration;
    void delay(){...}     // EFF: this / this
}

class Worker<w1>{
    TaskList<this, this> tasks;
    void add(Task<this & ?> t){...}
```

[6] Through the lookup function we skip the requirement for the definition of syntax.

Effects of expressions

$$\frac{q \in \{\mathtt{this}, x\}}{\Gamma \vdash_{\mathsf{e}} q \,:\, \epsilon \;/\; \epsilon} \qquad \frac{\Gamma \vdash t}{\Gamma \vdash_{\mathsf{e}} \mathtt{new}\ t \,:\, \epsilon \;/\; \epsilon} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi_1 \;/\; \phi_2 \quad \Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_3 \quad \Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_4 \qquad \Gamma \vdash \phi_3 \ll_{\mathsf{e}} \phi_4}{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi_3 \;/\; \phi_4}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} q \,:\, \phi \;/\; \phi'}{\Gamma \vdash_{\mathsf{e}} q.f \,:\, \phi \cup q \;/\; \phi'} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi \;/\; \phi' \qquad \Gamma \vdash e : c < Q, \overline{Q} >}{\Gamma \vdash_{\mathsf{e}} e.f \,:\, \phi \cup Q \;/\; \phi'}$$

$$\frac{\Gamma \vdash_{\mathsf{e}} q \,:\, \phi_1 \;/\; \phi_2 \qquad \Gamma \vdash_{\mathsf{e}} e' \,:\, \phi_3 \;/\; \phi_4}{\Gamma \vdash_{\mathsf{e}} q.f = e' \,:\, \phi_1 \cup \phi_3 \cup q \;/\; \phi_2 \cup \phi_4 \cup q} \qquad \frac{\Gamma \vdash_{\mathsf{e}} e \,:\, \phi_1 \;/\; \phi_2 \qquad \Gamma \vdash_{\mathsf{e}} e' \,:\, \phi_3 \;/\; \phi_4 \qquad \Gamma \vdash e : c < Q, \overline{Q} >}{\Gamma \vdash_{\mathsf{e}} e.f = e' \,:\, \phi_1 \cup \phi_3 \cup Q \;/\; \phi_2 \cup \phi_4 \cup Q}$$

$$\frac{\phi/\phi' = \mathcal{M}_{\mathit{eff}}(c{<}\overline{p}{>}, m) \qquad \Gamma \vdash_{\mathsf{e}} q \,:\, \phi_1 \;/\; \phi_2 \qquad \Gamma \vdash_{\mathsf{e}} e' \,:\, \phi_3 \;/\; \phi_4 \qquad \Gamma \vdash e' : \mathtt{[}\overline{Q/p}\mathtt{]}\, t'}{\Gamma \vdash_{\mathsf{e}} q.m(e') \,:\, \phi_1 \cup \phi_3 \cup q \cup \mathtt{[}\overline{Q/p}\mathtt{]}(\mathtt{[}q/\mathtt{this}\mathtt{]}\phi) \;\;/\;\; \phi_2 \cup \phi_4 \cup \mathtt{[}\overline{Q/p}\mathtt{]}\phi'}$$

$$\frac{\Gamma \vdash e : c{<}\overline{Q}{>} \qquad \phi/\phi' = \mathcal{M}_{\mathit{eff}}(c{<}\overline{p}{>}, m) \qquad \Gamma \vdash_{\mathsf{e}} e \,:\, \phi_1 \;/\; \phi_2 \qquad \Gamma \vdash_{\mathsf{e}} e' \,:\, \phi_3 \;/\; \phi_4 \qquad \Gamma \vdash e' : \mathtt{[}\overline{Q/p}\mathtt{]}\, t'}{\Gamma \vdash_{\mathsf{e}} e.m(e') \,:\, \phi_1 \cup \phi_3 \cup Q_{\mathit{1}} \cup \mathtt{[}\overline{Q/p}\mathtt{]}(\mathtt{[}Q_{\mathit{1}}/\mathtt{this}\mathtt{]}\phi) \;\;/\;\; \phi_2 \cup \phi_4 \cup \mathtt{[}\overline{Q/p}\mathtt{]}\phi'}$$

Effects inside other effects

$$\frac{}{\Gamma \vdash \epsilon \ll_{\mathsf{e}} \phi} \qquad \frac{\Gamma \vdash q \;\ll\; q'}{\Gamma \vdash q \ll_{\mathsf{e}} q'} \qquad \frac{\Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_2 \qquad \Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_3}{\Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_3} \qquad \frac{\Gamma \vdash \phi_1 \ll_{\mathsf{e}} \phi_3 \qquad \Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_4}{\Gamma \vdash \phi_1 \,\cup\, \phi_2 \ll_{\mathsf{e}} \phi_3 \,\cup\, \phi_4 \;\cup\; \phi_5}$$

Disjoint effects

$$\frac{}{\Gamma \vdash \epsilon \,\#\, \phi} \qquad \frac{\Gamma \vdash \phi \,\#\, \phi'}{\Gamma \vdash \phi' \,\#\, \phi} \qquad \frac{\begin{array}{c}\Gamma \vdash \phi \,\#\, \phi' \\ \Gamma \vdash \phi \,\#\, \phi''\end{array}}{\Gamma \vdash \phi \,\#\, \phi' \;\cup\; \phi''} \qquad \frac{\Gamma \vdash q \mathrel{\substack{\circ\\\circ}} q'}{\Gamma \vdash \; q \cap Q \,\#\, q' \cap Q'}$$

Figure 10. Effect rules for expressions, 'inside' and disjointness relations for effects.

```
                        // EFF: this / this
    void delay( ){...}  // EFF: this / this
}

class TaskList<l1,l2>{
    TaskList<l1,l2> next;
    Task<l2 & ?>  task;
    void add(Task<l2 & ?> t)   // EFF: l1 / l1
    void delay( ){...}         // EFF: l2 / l2
}

class Project<p>{ //
    TaskList<this, this> tasks;
    void add(Task<this & ?> t){...} //
                        // EFF: this / this
    void delay( ){...}  // EFF: this / this
}
```

4.2 Effects of Expressions

In this and the following sections we introduce effects for expressions and the disjointness and inside relations for effects. We go on to prove soundness of the effect system (theorem 3): that is, if the effects of two expressions are disjoint, then the order of their execution is unimportant.

The effects of expressions are defined through the judgment $\Gamma \vdash_{\mathsf{e}} e \;:\; \phi \;/\; \phi'$, given in figure 10. The rules are fairly straightforward, with effects of subexpressions propagated to the enclosing expression; reading or writing a field causing a read or write effect. Method invocation is more interesting: care must be taken to substitute the owners of the receiver into the effects of the method body correctly. The order of substitutions is crucial, as it was for the type rules. Furthermore, if the receiver of a field read, field write or method call is a path (*i.e.* a q), then the effect can be calculated more precisely. In our example:

```
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

final Project<this> p1=new Project<this>;
p1.delay();      // EFF: p1 / p1
```

The inside relation for effects ($\ll_{\mathsf{e}}$) is given in figure 10; one effect is inside another if it covers a smaller part of the heap.

4.3 Well-formed Programs with Effects

A program is well formed if, in addition to the requirements from section 3.7, a) the read/write effect of a method body is inside its declared effect, b) the declared write effect of a method body is within its declared read effect, and c) the effect of an overriding method is inside the effect of any overridden method. Formally, we require that a) $t\, m(t'\, x)\{e\}$ in $c{<}\overline{p}{>}$ implies that $\Gamma \vdash_{eff}\ e : \mathcal{M}_{\mathit{eff}}(c, m)$, b) $\mathcal{M}_{\mathit{eff}}(c, m) = \phi_1/\phi_2$ implies that $\Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi_1$, where $\Gamma = \overline{p}, c{<}\overline{p}{>}$ `this`, and c) $\mathcal{M}_{\mathit{eff}}(c, m) = \phi_1/\phi_2$ and $\mathcal{M}_{\mathit{eff}}(c', m) = \phi_3/\phi_4$ and $c{<}\overline{p}{>} \lhd c'{<}\overline{Q}{>}$ implies that $\Gamma \vdash \phi_1 \ll_{\mathsf{e}} [\overline{Q}/\overline{p}]\phi_3$ and $\Gamma \vdash \phi_2 \ll_{\mathsf{e}} [\overline{Q}/\overline{p}]\phi_4$, where $\Gamma = \overline{p}, c{<}\overline{p}{>}$ `this`.

As a counterpart to lemma 8, lemma 12 guarantees that the effect of an expression is preserved in a subclass modulo the necessary renamings for ownership parameters:

Definition 4. *For environment Γ, classes c and c', where $\overline{p} = \mathcal{D}m(c)$, $\overline{p'} = \mathcal{D}m(c')$, and effect ϕ, we define:*

$$\phi^{[c \lhd c']} = \begin{cases} [\overline{Q}/\overline{p}]\phi, & \text{if } c'{<}\overline{p'}{>} <:\ c{<}\overline{Q}{>} \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Lemma 12. *For classes c, and c', and environments Γ so that $\Gamma^{[c \lhd c']}$ is defined:*

- $\Gamma \vdash \phi \ll_{\mathsf{e}} \phi'$ *implies* $\Gamma^{[c\lhd c']} \vdash \phi^{[c\lhd c']} \ll_{\mathsf{e}} \phi'^{[c\lhd c']}$.
- $\Gamma \vdash_{\mathsf{e}} e : \phi \;/\; \phi'$ *implies* $\Gamma^{[c\lhd c']} \vdash_{\mathsf{e}} e^{[c\lhd c']} : \phi^{[c\lhd c']} \;/\; \phi'^{[c\lhd c']}$.

In well-formed programs, the write effect is always inside the read effect for any expression:

Lemma 13. *In a well-formed program, if $\Gamma \vdash_{\mathsf{e}} e : \phi \;/\; \phi'$, then $\Gamma \vdash \phi' \ll_{\mathsf{e}} \phi$.*

Proof. Straightforward induction on $\Gamma \vdash_{\mathsf{e}} e : \phi \;/\; \phi'$.

4.4 Projecting Effects onto the Heap

Based on the $\ll$ relation for objects (from figure 6), we define $[\![\, \phi \,]\!]_h$, the projection of an effect ϕ to a heap:

Definition 5.

$$\begin{array}{lcl} [\![\, r \,]\!]_h & = & \{\ \iota \mid h \vdash \iota \ll r\} \\ [\![\, r \cap R \,]\!]_h & = & [\![\, r \,]\!]_h \ \cap\ [\![\, R \,]\!]_h \\ [\![\, \phi\ \cup\ \phi' \,]\!]_h & = & [\![\, \phi \,]\!]_h\ \cup\ [\![\, \phi' \,]\!]_h \end{array}$$

We can prove that the type of an expression describes the boxes to which its evaluation will belong:

Lemma 14. *If $h \vdash e : c < R, \overline{R} >$ and $e, h \rightsquigarrow \iota, h'$, then $\iota \in [\![\, R \,]\!]_{h'}$.*

Proof. Straightforward application of the definitions (def 5, and $\ll$ from fig. 6), and theorem 1.

We give rules for judging the disjointness relation ($\Gamma \vdash \phi \,\#\, \phi'$) in figure 10. The rules state that the empty effect is disjoint from all effects; that the disjoint relation is symmetric and distributive with respect to the union of effects; and that if any pair of owners in a pair of sets of multiple owners are disjoint (by the $\between$ relation), then the effects denoted by this pair of sets is disjoint (by the $\#$ relation).

In the following lemma, the first two assertion guarantee soundness of the inside and disjointness judgments are sound wrt. the projection of effects. The last assertion is the counterpart to lemma 7.

Lemma 15. *For any effects ϕ, ϕ', environment Γ, substitution σ with $\Gamma, h \vdash \sigma$, and $\vdash h$, we have*

- *If* $\Gamma \vdash \phi \ll_{\mathsf{e}} \phi'$, *then* $[\![\, \sigma_h \circ \phi \,]\!]_h \subseteq [\![\, \sigma_h \circ \phi' \,]\!]_h$.
- *If* $\Gamma \vdash \phi \,\#\, \phi'$, *then* $[\![\, \sigma_h \circ \phi \,]\!]_h \cap [\![\, \sigma_h \circ \phi' \,]\!]_h = \emptyset$.
- *If* $\Gamma \vdash_{\mathsf{e}} e : \phi \;/\; \phi'$, *then* $h \vdash_{\mathsf{e}} \sigma_h \circ e : \sigma_h \circ \phi \;/\; \sigma_h \circ \phi'$.

Proof. by induction on derivations of $\Gamma \vdash \phi \ll_{\mathsf{e}} \phi'$, resp. $\Gamma \vdash \phi \,\#\, \phi'$, resp. $\Gamma \vdash_{\mathsf{e}} e : \phi \;/\; \phi'$, and using lemma 7.

4.5 Soundness of the Effects System

Soundness of the effects system guarantees that the read and write effects completely describe the areas of the heap read and written during some execution. We use the $*$ operator, inspired by separation logic notation, for "concatenation" of functions with disjoint domains. Thus, the construction $h * h'$ implicitly guarantees disjointness of h and h'. The notation $h|_A$ means the restriction of the mapping h to the domain A.

Theorem 2. *In a well formed program, if $\Gamma, h \vdash \sigma$, and $\Gamma \vdash_{\mathsf{e}} e : \phi \;/\; \phi'$, and $\sigma \circ e, h \rightsquigarrow \iota, h'$, then there exist heaps h_1, h_2, h_3, h_4 and h'_2 so that:*

- $h = h_1 * h_2 * h_3$, *and* $h' = h_1 * h'_2 * h_3 * h_4$,
- $e, h_1 * h_2 \rightsquigarrow \iota, h_1 * h'_2 * h_4$,
- $h_1 * h_2 = h|_{[\![\, \sigma_h \circ \phi \,]\!]_h}$ *and* $h_2 = h|_{[\![\, \sigma_h \circ \phi' \,]\!]_h}$
- $h_1 * h'_2 = h'|_{[\![\, \sigma_h \circ \phi \,]\!]_{h'}}$ *and* $h'_2 = h'|_{[\![\, \sigma_h \circ \phi' \,]\!]_{h'}}$

Proof. By induction on the derivation of $e, h \rightsquigarrow \iota, h'$. We use an "effects inversion lemma", *e.g.* $\Gamma \vdash_{\mathsf{e}} e.f : \phi \;/\; \phi'$ implies that $\Gamma \vdash_{\mathsf{e}} e : \phi_1 \;/\; \phi_2$, and $\Gamma \vdash e : c{<}Q, \overline{Q}{>}$, and $\Gamma \vdash \phi_1 \cup Q \ll_{\mathsf{e}} \phi$, and $\Gamma \vdash \phi_2 \ll_{\mathsf{e}} \phi'$, and $\Gamma \vdash \phi' \ll_{\mathsf{e}} \phi$ for some ϕ_1 and ϕ_2. We also use the fact that $e, h \rightsquigarrow \iota, h'$ implies that if h' and h'' are disjoint, then $e, h * h'' \rightsquigarrow \iota, h' * h''$.

We now prove that the execution of two expressions with disjoint effects is independent, in the sense that the order of their execution is immaterial:

Theorem 3. *In a well formed program, if $\Gamma, h \vdash \sigma$, and $\vdash h$ and $\Gamma \vdash_{\mathsf{e}} e_1 : \phi_1 \;/\; \phi_2$, and $\Gamma \vdash_{\mathsf{e}} e_2 : \phi_3 \;/\; \phi_4$, and $\Gamma \vdash \phi_1 \# \phi_4$ and $\Gamma \vdash \phi_2 \# \phi_3$, then*

$$\sigma \circ e_1, h \rightsquigarrow \iota', h'', \quad \sigma \circ e_2, h'' \rightsquigarrow \iota, h', \\ \textit{implies} \\ \sigma \circ e_2, h \rightsquigarrow \iota, h''', \;\; \sigma \circ e_1, h''' \rightsquigarrow \iota', h'$$

Proof. The proof is based on Matthew Smith's thesis [47], which develops an abstract model of independence of expressions based on disjointness of effects for any languages satisfying a set of basic requirements. Theorem 3.5.2 from [47] guarantees the assertion of our theorem provided that the heap satisfies basic composition and decomposition properties (**SH1-SH6** in [47]), that execution also satisfies basic decomposition properties (**LL2**,**L1-L5** in [47]), and that effects also satisfy decomposition properties (**LS1-LS5**). Property **LS4** corresponds to theorem 2. All the other properties can be easily proven for MOJO.

In terms of our example

```
w1 disjoint w2;
w1.delay();   // EFF: w1 & ? / w1 & ?
w2.delay();   // EFF: w2 & ? / w2 & ?
p1.delay();   // EFF: p1 & ? / p1 & ?
```

From `e1#e2` we obtain that `e1&?#e2&?` and therefore `e1.delay()` and `e2.delay()` are independent of each other in the sense of the above theorem. On the other hand, `e1.delay()` and `p1.delay()` are not necessarily independent as we have no information regarding the disjointness of `w1` and `p1`.

5. Discussion and Future Work

We plan to implement MOJO, investigate its applicability, especially extensions to support race-free programs and atomicity [9, 20]. In this section we discuss the repercussions of giving up owners as dominators, outline some idioms of multiple ownership, and some shortcomings in our use of `?`.

5.1 Giving up Owners as Dominators

As we said earlier, MOJO does not attempt to enforce the owners as dominators discipline. In other words, MOJO is a descriptive system: ownership *characterizes* the topology of the heap, rather than *constrains* it.

This is why MOJO does not require that ownership parameters preserve some "inside" relationship. Furthermore, without the owners as dominators discipline, and with the use of paths as actual owner parameters, some idioms, *e.g.* multiple iterators over one list, are straightforward to implement, *i.e.*

```
class List<l1,l2>{
    Node<this,l2> head;
    Iterator<this,l2> makeIterator()
        { new Iterator<this,l2>.next =  head; }
    ...
}
class Node<n1, n2> {
    Data<n2> d; Node<n1,n2> next; ...
}
class Iterator<i1,i2>{
    Node<i1,i2> next; ...
}
 ...
    final List<o1,o2> list1;
    Iterator<list1,o2> iter1 = list1.makeIterator();
```

In our example, `iter1.next` points to a node owned by `list1`. Note that we did *not* make use of multiple ownership, since all the nodes pointed at by *one* iterator belong to the *same* list.

In contrast, owners-as-dominators systems [7, 23, 5]) impose topological restrictions on heaps: a box's owner must be a dominator on all paths leading into the objects in the box: there can be no incoming pointers into a box (except from the box's owner). This amounts to requiring that

$$a \longrightarrow b \Longrightarrow a \in [\![\, \mathit{owner}(b) \,]\!]$$

that is, if a points to b, then a is inside b's owner.

We want to extend the MOJO type system so that references are permitted only if they come from within *one of the owners* (note we say $a \in \mathit{owners}(b)$ rather than $a \in \mathit{owner}(b)$). Thus, owners as dominators will apply to types instantiated with a single owner, and will be extended to owners as articulation points otherwise.

Furthermore, owners-as-modifiers systems, *e.g.* [31, 18], allow incoming pointers but forbid incoming messages that may modify an object: all modifications must pass via an object's owner. To represent owners-as-modifiers in MOJO, one would allow non-pure method calls only if the sender is inside the receiver's owners' boxes.

5.2 Idiom 1: Boxes for Variables

In contrast to many effects systems, *e.g.* OOFX [22], MOJO does not directly distinguish between object fields. For example, if `Task` had methods `delay` and `bribe` updating field `cost` and `time` respectively, then these methods would have effect "`this / this`" and MOJO would be unable to deduce their noninterference.

Field effects, although not directly included in our formal system, can be modelled with a simple idiom: Define a `IntBox` class with a single field, and getter and

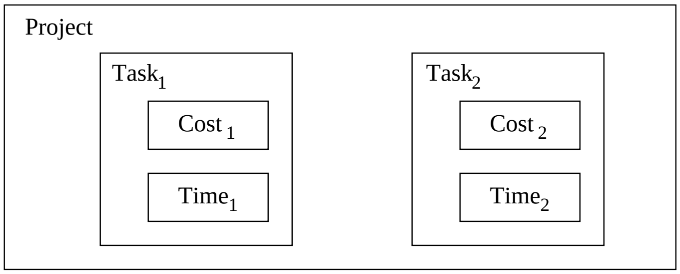

Figure 11. Cost and Time boxes inside Task boxes

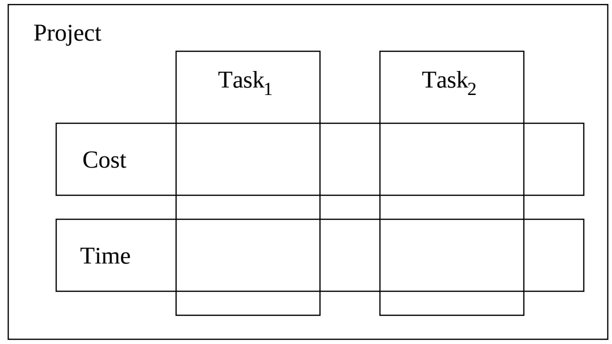

Figure 12. Nested, interlocking ownership.

setter methods for that field, affecting only that object. Rewrite the `Task` class to store each field in an `IntBox` with its own separate final field owner (see Figure 11), and use the getter and setter methods.

The effect of each method is localised to its `IntBox`, so wherever `IntBox`es are visible, the methods can be distinguished.

5.3 Idiom 2: Multiple Boxes per Object

In some cases it is useful to link boxes between nested objects. For example, in figure 12 the project has cost and duration boxes. We require each task's cost to be inside the project's cost, and similarly duration inside the project's duration. That would allows us to show that `delay` methods on projects affect only durations, and do not affect the cost of the project or any of its tasks (and vice versa).

We can code this by giving two additional ownership parameters to `Task` (*e.g.* `timeO` representing time, `costO` representing cost), and placing the time and cost objects into the intersection of the respective ownership boxes, *e.g.* through `Duration<this & timeO> time`.

5.4 Generics, and the meaning of `?`

An obvious extension of MOJO would be the introduction of generic types, so as, *e.g.* to allow the definition of generic lists [18, 42].

Another challenge is a more powerful notion of existential quantification than our current `?`, which is, we believe, adequate but could be more powerful. In particular, in our current solution, the `TaskList` is aware that its tasks have two owners, and the second one is unknown to the list. Thus, a `TaskList` whose tasks have three owners would require the declaration of a further class. This clearly restricts the reuse of the classes. It would be better if only classes `Worker` and `Project` were aware of the possible other owners of the tasks involved, and class `TaskList` was unaware of that. We plan to extend our approach so as to address this issue.

Seen from a related viewpoint, `?` is related to the notion of existential types. Thus, a list of tasks which share the same hidden owner would be `∃X.List<Task<X>>` while a list of tasks where each tasks has a potentially different, hidden owner would be `List<∃X.Task<X>>`. Note that the counterpart to the former is expressible but not denotable using Java wildcards [11].

6. Related Work

MOJO draws on two primary sources — on effects systems and on ownership types — and more recently, on work combining the two. Larger surveys of these areas can be found in [15, 41, 47]; here we provide an overview.

Effects systems and other approaches for syntactic control of interference have been developed for over thirty years [33, 44]. After interesting precursor work by Daniel Jackson [27], work on effects systems for object-oriented programs began with Leino's Data Groups [30] and Greenhouse and Boyland's Object-Oriented Effects System (OOFX) [22]. Data Groups were designed to support framing of changes across inheritance hierarchy, while OOFX provides a more general framework for reasoning about object-oriented programs.

Ownership types [14] were created by Clarke [12] to implement the flexible alias protection proposal [38]. Several variants of ownership types have been built including Confined Types [7, 23], Ownership Domains [2, 28], Generic Ownership [41, 42], Universes [35, 18], and have been used for purposes ranging from program verification [29, 36] to concurrency [8], to real-time memory management [5].

While these systems vary in the provided language constructs, type systems, and invariants, they all maintain the key constraint that every object has one owner at any given time. While some precursor work speculates about shared ownership, ours is the first to provide *multiple* owners.

The first system to combine effects and some form of ownership was Greenhouse and Boyland's OOFX [22]: effects from encapsulated subcomponents could be incorporated into effects upon their owners provided the subcomponents were accessed via a unique pointer. This system has recently been proven correct using adoption and separation logic [10]. OOFX includes a restricted form of multiple ownership in that instance regions can simultaneously belong to the instance, and to a corresponding region of a superordinate object. OOFX boxes (regions) cannot otherwise overlap, even though in *e.g.* Data Groups one field could be in more than one group. Boyland argues that intersecting regions limit effect separation: however multiple ownership's intersection and disjointness constraints remove this problem by making the program's local ownership topology clear: computations will be independent if their effects are known to be disjoint.

Clarke and Drossopoulou's JOE combines ownership with effects [13]. Unlike OOFX, JOE does not provide regions for variables or data groups within objects; JOE effects describe objects from a particular depth inside their owners. Smith subsequently constructed an effects system for ownership domains [46].

Lu and Potter designed a number of interesting ownership type systems based on effects [31, 32]. Effective ownership provides "effect encapsulation" — enforcing an owners-as-modifiers discipline without any constraints on inter-object references. They have built on this work to describe how ownership and effects can model invalidating (and obligations to revalidate) objects' invariants.

Most recently, Clifton's MAO [15, 16] uses an ownership and effects system to manage interference in an aspect-oriented language. MAO's ownership model is static, similar to that of confined types, with a set of global domains, generally one per aspect instance plus one for the base program. MAO's model is sufficient to detect aspect interference, and can be modelled in MOJO as a series of "global" boxes (a larger scale version of idiom 2 from section 5.3).

More generally, Multiple Ownership is related to other approaches to managing objects, effects, and allocation, such as region-based memory management [48] and alias types [45]. Multiple Ownership is also related to separation logic, in particular, Parkinson and Bierman's abstract invariants [40] can be seen as defining regions in the heap, as well as giving invariants for those regions. The key difference is that separation logic formulæ implicitly define the regions to which they apply, whereas ownership (types or assertions) define regions explicitly and independently of any formulæ. Finally, our ownership diagrams are related to set diagrams used in OO modelling, *e.g.* Spider and Constraint diagrams [21].

7. Conclusion

> *. . . structures like the city, which do require overlapping sets within them, are nevertheless persistently conceived as trees.*
>
> Christopher Alexander, **A City is not a Tree** [3]

A city is not a tree, and neither is a program [6, 34, 43]. Multiple ownership does not impose an ownership tree onto the objects in a program: it allows DAGs, and places objects into boxes — sets — that may intersect or remain disjoint as best serves the program's design. Using this *objects in boxes* model for ownership, we show how multiple ownership can be described as a smooth generalisation of single ownership systems. We have incorporated multiple ownership into the MOJO programming language design, including an effect system, that we have proven sound.

Acknowledgments This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project; by the EPSRC DTA grants; by the EPSRC grant Practical Ownership Types for Objects and Aspect Programs, EP/D061644/1; by Microsoft Research Cambridge; by a gift from Microsoft Research; and by the Royal Society of New Zealand Marsden Fund.

We are grateful to David Cunningham and Alex Buckley for their insightful comments, and to the anonymous OOPSLA referees for their encouraging and useful feedback.

References

[1] Marwan Abi-Antoun and Jonathan Aldrich. Ownership domains in the real world. In *IWACO workshop at ECOOP*, 2007.

[2] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.

[3] Christopher Alexander. A city is not a tree. *Design*, (206), 1966.

[4] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, 1997.

[5] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time Java. In *ECOOP*, 2006.

[6] Gareth Baxter, Marcus R. Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan D. Tempero. Understanding the shape of Java software. In *OOPSLA*, 2006.

[7] Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA*, 1999.

[8] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, November 2002.

[9] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, pages 56–69, Tampa Bay, FL, USA, 2001.

[10] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL*, 2005.

[11] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an existential types model for Java with wildcards. In *FTfJP workshop at ECOOP*, 2007.

[12] Dave Clarke. *Object Ownership and Containment*. PhD thesis, UNSW, Australia, 2002.

[13] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *OOPSLA*, 2002.

[14] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.

[15] Curtis Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs*. PhD thesis, Iowa State, 2005.

[16] Curtis Clifton, Gary T. Leavens, and James Noble. Ownership and effects for more effective reasoning about aspects. In *ECOOP*, 2007.

[17] Gilles Deleuze and Félix Guattari. *A Thousand Plateaus: Capitalism and Schizophrenia*. U. Minnesota, 1987.

[18] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP*, 2007.

[19] Sophia Drossopoulou. The benefits of putting objects into boxes. ESOP, 2006. Invited Talk.

[20] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *TLDI*, 2005.

[21] Joseph Gil, John Howse, and Stuart Kent. Towards a formalization of constraint diagrams. In *HCC*, 2001.

[22] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP*, 1999.

[23] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. In *OOPSLA*, 2001.

[24] Ralf Hinze. *The Fun of Programming*, chapter Fun with Phantom Types, pages 245–262. Palgrave Macmillan, 2003.

[25] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.

[26] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, November 1999.

[27] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM ToSEM*, 4(2), 1995.

[28] Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages. In *PLDI*, 2005.

[29] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.

[30] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, 1998.

[31] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *POPL*, pages 359–371, 2006.

[32] Yi Lu and John Potter. Object invariants and effects. In *ECOOP*, 2007.

[33] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

[34] Nick Mitchell. The runtime structure of object ownership. In *ECOOP*, 2006.

[35] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[36] Peter Müller. Reasoning about object structures using ownership. In *Verified Software: Theories, Tools, Experiments*, LNCS. Springer-Verlag, 2007.

[37] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a model of encapsulation. In *IWACO workshop at ECOOP*, 2003.

[38] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, 1998.

[39] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[40] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.

[41] Alex Potanin. *Generic Ownership — A Practical Approach to Ownership and Confinement in OO Programming Languages*. PhD thesis, 2007.

[42] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *OOPSLA*, 2006.

[43] Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in object-oriented programs. *Communications of the ACM*, May 2005.

[44] John C. Reynolds. Syntactic control of interference. In *POPL*, 1978.

[45] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP*, 2000.

[46] Matthew Smith. Effects system for ownership domains. In *FTfJP workshop at ECOOP*, 2005.

[47] Matthew Smith. *A Model of Effects with an application to Ownership Types*. PhD thesis, Imperial College, 2007.

[48] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[49] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC '04*, pages 1289–1296, 2004.

[50] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *JOT*, 2007.