Mojojojo — More Ownership for Multiple Owners

Paley Li
Victoria University of Wellington
lipale@ecs.vuw.ac.nz

Nicholas Cameron
Victoria University of Wellington
ncameron@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Abstract

Traditional ownership types organise the heap into a tree. Ownership types can support memory management, real-time systems, concurrency, parallelism, and general purpose reasoning about programs. Trees, however, are too restrictive to describe many real programs, limiting the usability of conventional ownership systems. *Multiple ownership* organises the heap into a directed acyclic graph, rather than a tree. MOJO was the first language to support multiple ownership; it featured multiple owners per object, owner-wildcards, an effect system, and a simple 'owners as sets' model.

In this paper, we introduce Mojojojo, a successor to MOJO. Mojojojo is both simpler and more powerful than MOJO: supporting generics and existential types (which allows for more re-usable classes); a more expressive system of constraints for specifying topology, which are closely tied to the simple set-theoretic model; and a simpler formalisation. We contribute a thorough description of Mojojojo, its formalisation and soundness proof, and a discussion of how Mojojojo can be extended to incorporate prescriptive constraints, addressing the same goals as owners-as-dominators or owners-as-modifiers disciplines.

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

General Terms Languages, Theory

Keywords Ownership types, multiple ownership, existential types, permissions, effects

> *Now, if you'll excuse me, I, Mojojojo, have a town to take over. I have a world to conquer. I have to seize control of an area and force its inhabitants to follow my way of thinking.*[1]

1. Introduction

Ownership types are a static type system for enforcing a structure over the heap. This structure allows for sophisticated reasoning about the heap in a number of domains. Most ownership systems force the heap into a tree. Empirical studies of object-oriented programs [25, 24], however, have shown that runtime object structures are more complex than simple trees. Objects may be shared between multiple threads or domains, suffer root sharing, diamond sharing, mutual ownership (butterflies) — as well as forming clean arboreal hierarchies (trees).

Multiple ownership structures the heap as a directed acyclic graph (DAG), rather than a tree. Multiple ownership can therefore accommodate programs with more complex heap structures than traditional ownership systems. In a multiple ownership world, objects may be owned by more than one object. Multiple ownership is supported in the MOJO language [9]. But, MOJO has several problems: classes designed for multiple ownership cannot always have the same declarations as those designed for a single owner; types are more restrictive than necessary; the language does not support encapsulation policies by restricting inter-object references or modifications; and the formalization is untidy.

In this paper, we describe and formalize a multiple ownership system, Mojojojo, that attempts to resolve these problems. The overarching design goal of Mojojojo has been to unify as many features as possible to produce a simple and general language. We have tried to produce a system that is expressive and flexible, but is also simple and re-usable. We use a system of constraints, based straightforwardly on the rules of set algebra, to describe heap topologies. We use generics and existential quantification to describe these topologies in types, which allows a single class to describe objects which may be singly or multiply owned (this was not possible in MOJO). Finally, Mojojojo has a smaller and less complex formal foundation than MOJO, and so should be easier to combine with other ownership type systems or to incorporate into programming languages.

We make the following contributions:

- formal and informal descriptions of a multiple ownership system, Mojojojo, which is polymorphic over all combinations of known and unknown, and single and multiple owners;
- a general, set-theoretic constraint system to describe ownership topologies;
- a type soundness proof for Mojojojo;
- the sketch of dual systems of permissions and effects that describe the permitted computations (permissions) and describe the consequences of computation (effects).

The rest of this paper is organized as follows: in Sect. 2 we introduce ownership types, multiple ownership, and existential ownership; in Sect. 3 we introduce the main features of our new language, Mojojojo; in Sect. 4 we describe these features formally, and describe some of the properties of the system, including type soundness; in Sect. 5 we introduce our work towards permissions and effects; in Sect. 6 we discuss how our system might be used in a real language and future work; in Sect. 7 we discuss related work; finally, in Sect. 8 we conclude.

[1] This and other quotes adapted from [1].

FOOL '10 Reno,Nevada.

2. Background

Ownership types [12, 27] are designed to make objects' runtime topologies explicit in programs, allowing programmers to describe and control their programs' runtime object topologies directly. For example, an ownership type `o:C` denotes an object of class `C` with *owner* `o`. The owner `this` means that objects are owned by the current instance of a class (i.e. "`this`"). The owner `owner` means the object that owns `this`. Owners can be generic in much the same way as types [29]. For example we can write:

```
class List<E> {
 this:Array<E> elems;
 int used = 0;
 void add(E elt) {
   if (used > elems.length)
     { // throw out of memory error!  }
   elems[used] = elt;
   used++;
 }
}
```

to start declaring a generic List class backed by an array. The type `this:Array<E>` says that each List instance will own its own Array, but both the types and ownership of the elements in the array are polymorphic, specified by the parameter `E`.

The semantics of ownership types depend on the particular ownership scheme chosen. The earliest ownership type systems [12, 10] enforce an ownership invariant known as "owners-as-dominators": all paths from the roots of the system to any owned objects (like the array) must pass via that object's owner (the list in this case). This enforces an encapsulation discipline similar to the law of Demeter [19]: component parts of an object are encapsulated within their owners, and cannot be accessed outside.

Ownership systems designed to support verification of object invariants generally support a different invariant known as the "Universes" invariant [26, 14] or "owners-as-modifiers". In these systems, any object may refer to any other object, but objects can only be modified by their owners. Therefore the array in our List example can be read anywhere, but can only be modified within the owner of this particular instance of List.

Ownership types can be used to describe the scope of effects [11, 4, 9, 20]. An effect system [21, 16] describes the *side-effects* of executing code. Effects can improve reasoning about a program by the compiler, allowing for fine-grained parallelisation, re-ordering of expressions, and determining which object's invariants need to be re-established. Effect based ownership systems generally constrain neither inter-object references or modifications; rather, they allow compilers (or programmers) to show that two computations will not interfere with each other, by showing that their effects are disjoint.

2.1 Existential Ownership

Ownership types must be *invariant* in their owner component to ensure soundness. This is similar to parametric types, which must be invariant[2] to ensure soundness (for example, generics in Java [17] but not, infamously, arrays in early versions of Eiffel [13]). It is sometimes convenient, however, to have collections of objects belonging to different owners, and this requires variant ownership. As in the world of type parametricity [18, 8], this feature can be safely supported using existential types [7].

Existential quantification of owners allows the programmer to specify that an owner is unknown (for example, $\exists$`o.o:Record` denotes objects of class `Record` with an unknown owner). A variable with such an existential type can store objects with any owner, but the system is sound because once the owner is 'forgotten' (by storing in a variable with quantified type) it cannot be recovered (assignment back into a variable with a specific owner is forbidden). Bounds on quantified variables can be used to represent partial knowledge about owners.

2.2 Multiple Ownership and MOJO

Although ownership types bring many benefits, they are often too restrictive for common use. Much of the research on ownership types has focused on making more flexible systems; however, nearly all systems are limited to a hierarchical structure. This is not good enough for many programming patterns: Mitchell [24] found that "more than 75%" of his sample programs which had an ownership structure could not be described using a simple ownership tree. Multiple ownership [9] lifts the restriction of hierarchies: by allowing objects to have multiple owners, the heap can be described as a directed acyclic graph (DAG).

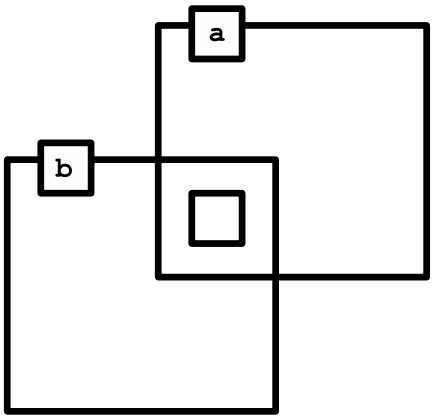

Figure 1. An object owned by two owners.

If an object is owned by multiple objects then we imagine that it is in the *intersection* (as in standard set theory) of those objects' contexts. We write `a`$\cap$`b:Record` for the type of records owned by `a` *and* `b`. This situation is represented in Fig. 1. Using such intersections, a sound multiple ownership graph can be used to denote effects in the same way as regular ownership. In MOJO, any number of contexts (including zero) may be intersected together to make a single context.

Here is an example which we will extend as we go along[3]:

```
class Doctor {
    this:List<this∩?:Record> recordList;
}

class Record {
    ?:Patient p;
    this:Object data;
}
```

A Doctor shares ownership over the records of the patients she treats with other unknown objects, possibly other doctors who treat each patient. In the `Record` class the owner of patient `p` is unknown and is represented with a wildcard (quantified context). Since a patient may be treated by one or many doctors and the other

[2] Languages can safely support variance with special mechanisms such as Scala's variance annotations or Java wildcards.

[3] The example uses MOJO's less expressive wildcard notation, rather than the explicit quantification we present in Mojojojo.

owners are unknown, a patient is owned by 'this' intersected with another wildcard. Given this type, the first two additions are legal, the second two illegal (neither include doc in the owner):

```
hospital:Doctor doc = new hospital:Doctor();
doc.recordList.add(new doc:Record());
doc.recordList.add(
    new (doc∩otherDoc∩anotherDoc):Record());
//doc.recordList.add(new ∅:Record());
//doc.recordList.add(
//  new (otherDoc∩anotherDoc):Record());
```

In order to support multiple ownership, the type system needs a little more information from the programmer about contexts. The programmer must specify which combinations of contexts may intersect, which are known to be disjoint, and which are neither. If two contexts are declared to intersect, then their intersection can be used as a context parameter; if two contexts are not known to intersect, then their intersection cannot be used as a context parameter. Disjointness information is used to calculate which effects are disjoint.

Although MOJO is a useful improvement over standard, single-owner ownership types it has several flaws: its support for variant types is restrictive and means that classes can not be agnostic as to whether they support single or multiple ownership, there is no support for ownership-based encapsulation, and the formalisation is complex.

3. An Informal Meeting with Mojojojo

The hobo formerly known as MOJO is no mo'. From this day forward, I shall be known as Mojojojo!

In this section we explain the key features of Mojojojo: flexible and expressive constraints on multiple-ownership topologies; and combining existential quantification of owners with type parametricity in the multiple ownership context.

3.1 Topological Constraints

A box is the abstraction of a group of objects. The heap is structured by putting all objects into boxes. A box may be associated with an object (the owner) or composed of other boxes. If a and b are variables in our program, we use a and b to denote the boxes they own. In Mojojojo (as in MOJO), we can write a ∩ b for the intersection of the two boxes; objects in the box a ∩ b are owned by both a and b. MOJO allowed the intersecting together of many boxes, we only allow the intersection of two boxes, but since each box may in turn be an intersection, we are as expressive (and closer to set algebra).

We also allow taking the union of boxes; e.g., a∪b or a∩(b∪c). In MOJO, unions are present in effects, but not in the source code. Our approach is therefore more expressive and more uniform than MOJO.

In our patient records example we could write a type such as:

```
doc1∪(doc2∪doc3):Record
```

to indicate the record for a patient treated by either of three doctors (perhaps in a busy emergency ward).

In Mojojojo, constraints express the relationships between boxes. In MOJO, the programmer could specify that boxes may overlap or were known to be disjoint. In Mojojojo we use constraint notation: a ∩ b ≠ ∅ and a ∩ b = ∅, respectively. Our constraints extend easily to unions and compound boxes. These constraints are constraints on the topology of boxes, not on objects; therefore, they do not guarantee that an object is present in any particular box, only that it is allowed. Mojojojo also allows the specification of constraints to describe whether boxes are inside or outside another box. These constraints are familiar from other ownership systems [10, 11]; in Mojojojo, they are described in terms of sub-boxing (c.f., subsets): a ⊆ b. Our constraint system is equipped with the usual rules of set algebra and therefore deals consistently with these different kinds of constraint.

We consider the very close relationship with standard set theory to be one of the important improvements of Mojojojo over MOJO and other ownership systems.

The relationship between ownership and set algebra based constraints is not always intuitive. For example, does a 'inside' b imply that a intersects b? In most ownership systems an object in a is not considered to be in b [10]. This suggests the intersection should be considered empty. However, our constraints describe the ownership topology itself, not how objects are located within this topology. Our topological rules follow set theory and the intersection of a and b, in this instance, is considered non-empty.

Similarly to other ownership systems, object ownership is invariant with respect to the topological inside relation; that is, an object in a is not also in b. Furthermore, an object inside a∩b is not also in b, independent of the relation between a and b.

3.2 Generics and Existential Owners

We expect that variant owners (owners which can vary with subtyping) will be used more under multiple ownership than single ownership. A common use is to refer to a multiply-owned object by a known and unknown owner, for example, this∩?:C (using MOJO-like syntax). In MOJO, these unknown owners are denoted with a wildcard ?. Unfortunately, this syntax has some drawbacks: classes cannot be polymorphic in the variance of their owners (for example, a list of objects with known owners must have a separate class definition from a list of objects with partially known (or unknown) owners [9]) and bounds cannot be given for ?[4].

In Jo∃ [7], existential quantification of owners is used to represent unknown and variant contexts. We adapt this quantification to multiple owners. In Mojojojo, we quantify boxes, so an existentially quantified owner might hide a combination of contexts. Furthermore, the inside/outside bounds on the traditional single ownership hierarchy are not flexible enough for the DAGs of multiple ownership; instead, we use constraints in existential types. Constraints take the place of bounds, describing where a quantified context appears in the DAG.

A benefit of existential quantification is that our formalisation is very clean. The well-known concepts [28] of packing and unpacking are used implicitly[5] to implement the restricted access to quantified variables. In MOJO, the same restrictions required a complex system of strict lookups and ad hoc type transformations (see [6] for a technical comparison of these approaches).

In our running example, we could add the following constraints to the existential type of the record to ensure that only objects which are inside the current object's owner may share in owning each record, and that these other objects must be allowed to overlap with the current object:

```
this:List<∃o;(o⊆owner,o∩this≠∅).(this∩o):Record>
   recordList;
```

We add generics (parametric types) to Mojojojo, not for the usual benefits of more precise and flexible types (although these are welcome), but because, by combining generics with existential

[4] The second problem could be remedied by allowing bounds on ? (as in Java wildcards).

[5] Implicit packing and unpacking follows recent formalisations of Java wildcards [8], and contrasts with explicit packing and unpacking in traditional existential types formalisations [23].

quantification, our classes are variance-polymorphic. The result of this is that class declarations for single and multiple ownership systems are identical and therefore classes written for single ownership can be reused in Mojojojo.

Given the `List` class declaration in Sect. 2, we can write the following types (in each case, the list itself is owned by `a`):

`a:List<this:Record>`	list of records owned by `this`
`∃o.a:List<o:Record>`	list of records owned by a single unknown owner
`a:List<∃o.o:Record>`	list of records owned by different owners

In MOJO, we would need two list classes, one which handled the first case and one which handled the third; the second case could not be encoded at all.

4. Formalisation

The way I communicate is much different. I do not reiterate, repeat, and reinstate the same thing over and over again. I am clear, concise, to the point!

In this section we present our formalisation of Mojojojo, a calculus in the tradition of Featherweight Java [17]. For the sake of simplicity (and since it is orthogonal to ownership), we do not support inheritance.

Q	::=	`class C<`$\overline{\texttt{X}}$ $\overline{\mathcal{C}}$`>` $\{\overline{T\ \texttt{f;}}\ \overline{M}\}$	*class declarations*
M	::=	T `m(`T `x)` $\overline{\mathcal{C}}$ `{return` e`;}`	*method declarations*
a	::=	γ | `world` | $\emptyset$ | T`.owner`	*contexts*
b	::=	a | $b \cap b$ | $b \cup b$	*boxes*
r	::=	ι | $r \cap r$ | $r \cup r$	*runtime boxes*
$\mathcal{C}$	::=	$b \subseteq b$ | $b = \emptyset$ | $b \neq \emptyset$	*constraints*
N	::=	b`:C<`$\overline{T}$`>`	*class types*
T	::=	$\exists\emptyset;\emptyset$`.X` | $\exists\overline{\texttt{o}};\overline{\mathcal{C}}.N$ | $\top$	*types*
R	::=	$\exists\emptyset;\emptyset.r$`:C<`$\overline{T}$`>`	*runtime types*
e	::=	`null` | γ | γ`.f` | γ`.f =` e | γ`.m(`e`)` | `new` b`:C<`$\overline{T}$`>` | `err`	*expressions*
v	::=	`null` | ι	*values*
γ	::=	`x` | ι	*expression variables and addresses*
Γ	::=	$\overline{\gamma:T}$	*environments*
Δ	::=	$\overline{\mathcal{C}}$	*constraint environments*
$\mathcal{H}$	::=	$\overline{\iota \rightarrow \{N,\ \overline{\texttt{f}\rightarrow v}\}}$	*heaps*
`x,o,owner,this`			*variables*
`X`			*type variables*
`f`			*field names*
`m`			*method names*
`C`			*class names*

Figure 2. Syntax of Mojojojo.

We present the syntax for Mojojojo in Fig. 2. Constraints ($\mathcal{C}$) in class declarations and existential types are associated with formal context variables (`o`). All class types are generic and existential. Non-existential types can be simulated by quantifying with the empty set of variables and constraints. Type variables (X) are always quantified by empty lists of variables and constraints; they are quantified for consistency with class types. We use a top type ($\top$) for variables that are only used as owners and are, therefore, not associated with a type. Syntax in grey cannot be written by the programmer and is used to represent running programs.

Our judgements are decided under three environments: Γ maps variables to their types, Δ stores the current constraints on contexts, and $\overline{\texttt{X}}$ is a list of type variables currently in scope (we do not support bounds on type variables). Variables in Γ may be either expression variables (`x`) or context variables (`o`, which always have type $\top$); the latter only appear as a result of unpacking existential types, and are not used in expressions. There is, however, no syntactic distinction between `x` and `o`. At runtime, Γ maps addresses (ι) to their types.

Mojojojo is not Turing complete, for example, conditionals cannot be encoded because we do not support inheritance or first class functions. Mojojojo is proposed as a formalisation of the interesting aspects of a multiple ownership language: we do not propose anyone try to program with it, therefore Turing completeness is not important. Type safety is an orthogonal question to Turing completeness, so is not affected.

Sub-boxes: $\boxed{\Delta;\Gamma;\overline{\texttt{X}} \vdash b \subseteq b}$

$$\frac{}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b \subseteq b}\ \text{(B-Reflex)} \qquad \frac{}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b \subseteq \texttt{world}}\ \text{(B-Top)}$$

$$\frac{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_2 \quad \Delta;\Gamma;\overline{\texttt{X}} \vdash b_2 \subseteq b_3}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_3}\ \text{(B-Trans)} \qquad \frac{b_1 \subseteq b_2 \in \Delta}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_2}\ \text{(B-Env)}$$

$$\frac{}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_1 \cup b_2}\ \text{(B-Join-1)} \qquad \frac{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_3 \quad \Delta;\Gamma;\overline{\texttt{X}} \vdash b_2 \subseteq b_3}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \cup b_2 \subseteq b_3}\ \text{(B-Join-2)}$$

$$\frac{}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \cap b_2 \subseteq b_1}\ \text{(B-Meet-1)} \qquad \frac{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_2 \quad \Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_3}{\Delta;\Gamma;\overline{\texttt{X}} \vdash b_1 \subseteq b_2 \cap b_3}\ \text{(B-Meet-2)}$$

$$\frac{\Gamma(\gamma) = b\texttt{:C<}\overline{\texttt{T}}\texttt{>}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma \subseteq b}\ \text{(B-Owner)} \qquad \frac{}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \emptyset \subseteq b}\ \text{(B-Empty)}$$

Equalities: $\boxed{b = b}$

$b_1 \cap b_2 = b_2 \cap b_1$	Eq-Comm-I
$b_1 \cup b_2 = b_2 \cup b_1$	Eq-Comm-U
$b_1 \cap (b_2 \cap b_3) = (b_1 \cap b_2) \cap b_3$	Eq-Assoc-I
$b_1 \cup (b_2 \cup b_3) = (b_1 \cup b_2) \cup b_3$	Eq-Assoc-U
$b_1 \cap (b_2 \cup b_3) = (b_1 \cap b_2) \cup (b_1 \cap b_3)$	Eq-Distrib-I
$b_1 \cup (b_2 \cap b_3) = (b_1 \cup b_2) \cap (b_1 \cup b_3)$	Eq-Distrib-U
$b \cap \texttt{world} = b$	Eq-Id-I
$b \cup \emptyset = b$	Eq-Id-U
$b \cap b = b$	Eq-Idem-I
$b \cup b = b$	Eq-Idem-U
$b \cap \emptyset = \emptyset$	Eq-Dom-I
$b \cup \texttt{world} = \texttt{world}$	Eq-Dom-U
$b_1 \cap (b_1 \cup b_2) = b_1$	Eq-Abs-I
$b_1 \cup (b_1 \cap b_2) = b_1$	Eq-Abs-U
$b\texttt{:C<}\overline{T}\texttt{>.owner} = b$	Eq-Owner

Figure 3. Mojojojo sub-boxing and equalities between boxes.

Valid constraints: $\boxed{\Delta;\Gamma;\overline{\mathtt{X}} \models \mathcal{C}}$

$$\frac{\mathcal{C} \in \Delta}{\Delta;\Gamma;\overline{\mathtt{X}} \models \mathcal{C}}\ \text{(C-ENV)} \qquad \frac{\Delta;\Gamma;\overline{\mathtt{X}} \models \mathcal{C}' \qquad \mathcal{C}' = \mathcal{C}}{\Delta;\Gamma;\overline{\mathtt{X}} \models \mathcal{C}}\ \text{(C-EQ)}$$

$$\frac{\Delta;\Gamma;\overline{\mathtt{X}} \models b' = \emptyset \quad \Delta;\Gamma;\overline{\mathtt{X}} \vdash b \subseteq b'}{\Delta;\Gamma;\overline{\mathtt{X}} \models b = \emptyset}\ \text{(C-SB-E)} \qquad \frac{\Delta;\Gamma;\overline{\mathtt{X}} \models b' \neq \emptyset \quad \Delta;\Gamma;\overline{\mathtt{X}} \vdash b' \subseteq b}{\Delta;\Gamma;\overline{\mathtt{X}} \models b \neq \emptyset}\ \text{(C-SB-NE)}$$

$$\frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b_1 \subseteq b_2}{\Delta;\Gamma;\overline{\mathtt{X}} \models b_1 \subseteq b_2}\ \text{(C-SB)}$$

Figure 4. Mojojojo valid constraints.

Subtyping: $\boxed{\Delta;\Gamma;\overline{\mathtt{X}} \vdash T <: T}$

$$\frac{}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash T <: T}\ \text{(S-REFLEX)} \qquad \frac{}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash T <: \top}\ \text{(S-TOP)}$$

$$\frac{\overline{\mathtt{o}} \cap fv(\exists \overline{\mathtt{o}'};\overline{\mathcal{C}'}.N) = \emptyset \quad \Delta,\overline{\mathcal{C}};\Gamma,\overline{\mathtt{o}:\top};\overline{\mathtt{X}} \models [\overline{b/\mathtt{o}'}]\overline{\mathcal{C}'}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \exists \overline{\mathtt{o}};\overline{\mathcal{C}}.[\overline{b/\mathtt{o}'}]N <: \exists \overline{\mathtt{o}'};\overline{\mathcal{C}'}.N}\ \text{(S-ENV)}$$

Figure 5. Mojojojo subtyping.

We define well-formedness judgements for Mojojojo in Fig. 6. These define well-formed types, contexts, constraints, and heaps. Each checks that variables are correctly bound, constraints are satisfied where appropriate (the equivalent of bounds checking), and named classes are declared in the program.

In Fig. 3 we define sub-box and equality relations between boxes. These rules follow directly from set theory, sub-boxing corresponds to the subset relation. We add rules B-ENV and B-OWNER to sub-boxing to take into account relations declared by the programmer, and that object's contexts are within their owner's context.

A key concept in Mojojojo is the constraint ($\mathcal{C}$). Constraints may be assumed within the body of the class, method, or unpacked scope, and must be satisfied when the class is instantiated, method invoked, or existential type packed.

Constraints are established to be valid using the rules in Fig. 4. These rules describe how topological constraints can be found valid using the sub-box and equality rules, declarations, and some intuitive notions about these constraints. Note that for a box to be judged equal or not equal to the empty set (e.g., $\Delta;\Gamma;\overline{\mathtt{X}} \models \mathtt{a} = \emptyset$) is a different concept than box-equality (e.g., $\mathtt{a} = \emptyset$). The former concept means that the constraint $\mathtt{a} = \emptyset$ can be proved under a given set of environments, the latter means that $\mathtt{a} = \emptyset$ under all environments, derived using the 'set equality' rules in Fig. 3. Also, a valid constraint is not necessarily well-formed, or vice-versa. Well-formedness is a purely syntactic judgement, which checks mainly that component variables are in scope; whereas validity is a semantic judgement which checks that the constraint can be proved from the relevant environments.

Subtyping is defined in Fig. 5; since we do not support subclassing, the only interesting part is handling existential types. We use an ENV rule, taken from formalisations of Java wildcards [22]. This rule allows introduction of existential types (packing), and partial abstraction of existential types (corresponding to co- and contravariant changes to bounds in traditional existential types systems). In Mojojojo, strictness of bounds is indicated by the set of constraints in the subtype being stronger than (that is, can prove valid) the set of constraints in the supertype. The *fv* function returns the set of free variables in a type.

Well-formed boxes: $\boxed{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b\ \text{OK}}$

$$\frac{\Gamma(\gamma) \neq \mathtt{X}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \gamma\ \text{OK}}\ \text{(F-VAR)} \qquad \frac{}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \mathtt{world}\ \text{OK}}\ \text{(F-WORLD)}$$

$$\frac{}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \emptyset\ \text{OK}}\ \text{(F-EMPTY)} \qquad \frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash T\ \text{OK}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash T\mathtt{.owner}\ \text{OK}}\ \text{(F-TOWNER)}$$

$$\frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b\ \text{OK} \quad \Delta;\Gamma;\overline{\mathtt{X}} \vdash b'\ \text{OK} \quad \Delta;\Gamma;\overline{\mathtt{X}} \models b \cap b' \neq \emptyset}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b \cap b'\ \text{OK}}\ \text{(F-INTERSECT)} \qquad \frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b\ \text{OK} \quad \Delta;\Gamma;\overline{\mathtt{X}} \vdash b'\ \text{OK}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b \cup b'\ \text{OK}}\ \text{(F-UNION)}$$

Well-formed types: $\boxed{\Delta;\Gamma;\overline{\mathtt{X}} \vdash T\ \text{OK}}$

$$\frac{\mathtt{X} \in \overline{\mathtt{X}}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \mathtt{X}\ \text{OK}}\ \text{(F-TYPE-VAR)} \qquad \frac{\Gamma' = \Gamma,\overline{\mathtt{o}:\top} \quad \Delta,\overline{\mathcal{C}};\Gamma';\overline{\mathtt{X}} \vdash N\ \text{OK} \quad \Delta,\overline{\mathcal{C}};\Gamma';\overline{\mathtt{X}} \vdash \overline{\mathcal{C}}\ \text{OK}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \exists \overline{\mathtt{o}};\overline{\mathcal{C}}.N\ \text{OK}}\ \text{(F-EXISTS)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{Y}}\ \overline{\mathcal{C}}\mathtt{>\ ...} \quad \Delta;\Gamma;\overline{\mathtt{X}} \vdash \overline{T}\ \text{OK} \qquad \Delta;\Gamma;\overline{\mathtt{X}} \vdash b\ \text{OK} \quad \Delta;\Gamma,\mathtt{this}{:}b{:}\mathtt{C<}\overline{T}\mathtt{>};\overline{\mathtt{X}} \models [b/\mathtt{owner},\overline{T/\mathtt{Y}}]\overline{\mathcal{C}}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b{:}\mathtt{C<}\overline{T}\mathtt{>}\ \text{OK}}\ \text{(F-CLASS)}$$

Well-formed constraints: $\boxed{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \mathcal{C}\ \text{OK}}$

$$\frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b\ \text{OK}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b = \emptyset\ \text{OK}}\ \text{(F-EQ)} \qquad \frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b\ \text{OK}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b \neq \emptyset\ \text{OK}}\ \text{(F-NEQ)}$$

$$\frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b_1\ \text{OK} \qquad \Delta;\Gamma;\overline{\mathtt{X}} \vdash b_2\ \text{OK}}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b_1 \subseteq b_2\ \text{OK}}\ \text{(F-SUB)}$$

Well-formed heap: $\boxed{\vdash \mathcal{H}\ \text{OK}}$

$$\frac{\mathcal{H} \vdash N\ \text{OK} \quad \dfrac{\forall \iota \to \{N;\overline{\mathtt{f}\to v}\} \in \mathcal{H} : fType(\mathtt{f},N) = T' \qquad \mathcal{H} \vdash v : [\iota/\mathtt{this}]T'}{\forall v \in \overline{v} : v \neq \mathtt{null} \Rightarrow v \in dom(\mathcal{H})}}{\vdash \mathcal{H}\ \text{OK}}\ \text{(F-HEAP)}$$

Figure 6. Mojojojo well-formed contexts, types, and heap.

Expression typing is defined in Fig. 8. In T-FIELD, T-ASSIGN, and T-INVK the type of the receiver is unpacked before performing field or method lookup. The resulting type may contain free variables, and if this type forms part of the conclusion, then it must be packed to prevent free variable escape. Packing is done using the $\Downarrow$ (close) operation (defined in Fig. 7): variables and constraints to be packed ($\overline{\mathtt{o}}$ and $\overline{\mathcal{C}}$ in $\Downarrow_{\overline{\mathtt{o}};\overline{\mathcal{C}}}$) are added to those quantifying a

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{> }\{\overline{U\ \texttt{f;}}\ \overline{M}\}}{fields(\texttt{C}) = \overline{\texttt{f}}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{> }\{\overline{U\ \texttt{f;}}\ \overline{M}\}}{fType(\texttt{f}_i, b\texttt{:C<}\overline{T}\texttt{>}) = [b/\texttt{owner}, \overline{T/\texttt{X}}]U_i}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{> }\{\overline{U\ \texttt{f;}}\ \overline{M}\} \\ T\ \texttt{m(}T'\ \texttt{x)}\ \overline{\mathcal{C}''}\ \{\texttt{return } e\texttt{;}\} \in \overline{M}\end{array}}{mBody(\texttt{m}, b\texttt{:C<}\overline{T}\texttt{>}) = (\texttt{x}; [b/\texttt{owner}, \overline{T/\texttt{X}}]e)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{> }\{\overline{U'\ \texttt{f;}}\ \overline{M}\} \\ T\ \texttt{m(}T'\ \texttt{x)}\ \overline{\mathcal{C}''}\ \{\texttt{return } e\texttt{;}\} \in \overline{M}\end{array}}{mType(\texttt{m}, b\texttt{:C<}\overline{U}\texttt{>}) = [b/\texttt{owner}, \overline{U/\texttt{X}}](\overline{\mathcal{C}''}.T' \to T)}$$

$$\frac{}{\Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}}\ \exists\emptyset;\emptyset\texttt{.X} = \exists\emptyset;\emptyset\texttt{.X}}$$

$$\frac{}{\Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}}\ \exists\overline{\texttt{o}'};\overline{\mathcal{C}'}.N = \exists\overline{\texttt{o}},\overline{\texttt{o}'};\overline{\mathcal{C}},\overline{\mathcal{C}'}.N}$$

$$\frac{}{\Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}}}\ \top = \top}$$

Figure 7. Auxiliary functions for Mojojojo.

Expression typing: $\boxed{\Delta;\Gamma;\overline{\texttt{X}} \vdash e : T}$

$$\frac{\Delta;\Gamma;\overline{\texttt{X}} \vdash N\ \text{OK}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \texttt{new } N : N}\ \text{(T-NEW)} \qquad \frac{}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma : \Gamma(\gamma)}\ \text{(T-VAR)}$$

$$\frac{\Delta;\Gamma;\overline{\texttt{X}} \vdash T\ \text{OK}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \texttt{null} : T}\ \text{(T-NULL)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma : \exists\overline{\texttt{o}};\overline{\mathcal{C}}.b\texttt{:C<}\overline{\texttt{T}}\texttt{>} \\ fType(\texttt{f}, b\texttt{:C<}\overline{\texttt{T}}\texttt{>}) = T\end{array}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma\texttt{.f} : \Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}},\gamma\subseteq b}\ [\gamma/\texttt{this}]T}\ \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma : \exists\overline{\texttt{o}};\overline{\mathcal{C}}.b\texttt{:C<}\overline{\texttt{T}}\texttt{>} \qquad fType(\texttt{f}, b\texttt{:C<}\overline{\texttt{T}}\texttt{>}) = T' \\ \Delta;\Gamma;\overline{\texttt{X}} \vdash e : T \qquad \Delta, \overline{\mathcal{C}}, \gamma \subseteq b; \Gamma, \overline{\texttt{o:}\top}; \overline{\texttt{X}} \vdash T <: [\gamma/\texttt{this}]T'\end{array}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma\texttt{.f = } e : T}\ \text{(T-ASSIGN)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma : \exists\overline{\texttt{o}};\overline{\mathcal{C}}.b\texttt{:C<}\overline{\texttt{T}}\texttt{>} \qquad \Delta;\Gamma;\overline{\texttt{X}} \vdash e : T'' \\ mType(\texttt{m}, b\texttt{:C<}\overline{\texttt{T}}\texttt{>}) = \overline{\mathcal{C}'}.T' \to T \\ \Delta, \overline{\mathcal{C}}, \gamma \subseteq b; \Gamma, \overline{\texttt{o:}\top}; \overline{\texttt{X}} \vdash T'' <: [\gamma/\texttt{this}]T' \\ \Delta, \overline{\mathcal{C}}, \gamma \subseteq b; \Gamma, \overline{\texttt{o:}\top}; \overline{\texttt{X}} \models [\gamma/\texttt{this}]\overline{\mathcal{C}'}\end{array}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash \gamma\texttt{.m(}e\texttt{)} : \Downarrow_{\overline{\texttt{o}};\overline{\mathcal{C}},\gamma\subseteq b}\ [\gamma/\texttt{this}]T}\ \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\overline{\texttt{X}} \vdash e : T' \\ \Delta;\Gamma;\overline{\texttt{X}} \vdash T' <: T \qquad \Delta;\Gamma;\overline{\texttt{X}} \vdash T\ \text{OK}\end{array}}{\Delta;\Gamma;\overline{\texttt{X}} \vdash e : T}\ \text{(T-SUBS)}$$

Figure 8. Mojojojo expression typing rules.

class type or ignored in the case of type variables (type variables have no free expression variables). In the premises of the type rules (intuitively, between unpacking and repacking) we must be careful to use correct, enlarged environments which include the unpacked context variables. Ownership types are dependent types and `this` can be used in types, constructing runtime types involves substituting the receiver for `this` in types, thus we restrict receviers to be variables. This is not a practicle problem because methods can be used like let expressions to assign any expression to a variable, which can then be used as receiver.

$$\frac{\begin{array}{c}\Gamma = \texttt{owner:}\top, \texttt{this : owner:C<}\overline{\texttt{X}}\texttt{>} \\ \texttt{this} \neq \emptyset, \overline{\mathcal{C}}; \Gamma; \overline{\texttt{X}} \vdash \overline{T}, \overline{M}\ \text{OK} \qquad \texttt{this} \neq \emptyset; \Gamma; \overline{\texttt{X}} \vdash \overline{\mathcal{C}}\ \text{OK} \\ \texttt{this} \neq \emptyset, \overline{\mathcal{C}}; \Gamma; \overline{\texttt{X}} \not\models \emptyset \neq \emptyset\end{array}}{\vdash \texttt{class C<}\overline{\texttt{X}}\ \overline{\mathcal{C}}\texttt{> }\{\overline{T\ \texttt{f;}}\ \overline{M}\}\ \text{OK}}\ \text{(T-CLASS)}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \texttt{x:}T' \qquad \Delta' = \Delta, \overline{\mathcal{C}} \\ \Delta'; \Gamma; \overline{\texttt{X}} \vdash T, T'\ \text{OK} \qquad \Delta'; \Gamma'; \overline{\texttt{X}} \vdash e : T \\ \Delta'\Gamma; \overline{\texttt{X}} \not\models \emptyset \neq \emptyset \qquad \Delta; \Gamma; \overline{\texttt{X}} \vdash \overline{\mathcal{C}}\ \text{OK}\end{array}}{\Delta; \Gamma; \overline{\texttt{X}} \vdash T\texttt{m(}T'\ \texttt{x)}\ \overline{\mathcal{C}}\ \{\texttt{return } e\texttt{;}\}\ \text{OK}}\ \text{(T-METHOD)}$$

Figure 9. Mojojojo typing rules for classes and methods.

Type rules for methods and classes are given in Fig. 9. Declared constraints must be consistent as well as being well-formed. Consistency is checked by requiring that the empty set cannot be proved not equal to itself (in both T-CLASS and T-METHOD); essentially 'false' cannot be proved.

Mojojojo has a standard, large step semantics; we relegate it, and rules for type checking at runtime; to the appendix. We use large steps because in the version of Mojojojo with permissions and effects we need to model a stack, this is not needed in the presentation in this paper, but we keep the same semantics.

4.1 Properties of Mojojojo

We have proved the subject reduction theorem for Mojojojo:

> **Theorem:** subject reduction
> *For all $\mathcal{H}$, $\mathcal{H}'$, e, v, and T, if $\mathcal{H} \vdash e : T$ and $e; \mathcal{H} \rightsquigarrow v; \mathcal{H}'$ and $\vdash \mathcal{H}$ OK then $\mathcal{H}' \vdash v : T$ and $\vdash \mathcal{H}'$ OK.*

The proof proceeds in a standard manner by structural induction over the derivation of reductions. We have a large number of lemmas, most of them standard for such systems (see for example, [9, 6, 7]). We state and discuss some of the interesting lemmas below. Subject reduction shows that reduction preserves types, that is an expression cannot change type as it executes (upto subtyping). In ownership terms, since we have proved preservation, we can be sure that the owners described by the static types in our language reflect the actual heap at runtime. This is a necessary condition for supporting an effect system or any prescriptive property. Rules for using the heap as an environment (thereby allowing the static type rules to judge runtime expressions) are given in the appendix.

> **Lemma:** strengthening (type checking) *For all Δ, Γ, $\overline{\texttt{X}}$, e, T, and $\overline{\mathcal{C}}$, if $\Delta, \overline{\mathcal{C}}; \Gamma; \overline{\texttt{X}} \vdash e : T$ and $\Delta; \Gamma; \overline{\texttt{X}} \models \overline{\mathcal{C}}$ then $\Delta; \Gamma; \overline{\texttt{X}} \vdash e : T$*

We use a system of constraints rather than bounds on variables, this causes several differences in the statement of lemmas (see below) and the overall shape of our proofs. One of the nice properties of our system is strengthening: if a constraint can be proved by an environment, then it can be removed from that environment when proving judgements. We give the strengthening lemma for the type checking judgement above, lemmas for the other judgements are similar.

> **Lemma:** box substitution preserves type checking *For all Δ, Γ, $\overline{\texttt{X}}$, e, T, $\overline{b}$, $\overline{\texttt{o}}$, and $\overline{\mathcal{C}}$, if $\Delta, \overline{\mathcal{C}}; \Gamma, \overline{\texttt{o:}\top}; \overline{\texttt{X}} \vdash e :$*

T and $\Delta;\Gamma;\overline{\mathtt{X}} \vdash \overline{b}$ OK and $\Delta;\Gamma;\overline{\mathtt{X}} \models [\overline{b/\mathtt{o}}]\overline{\mathcal{C}}$ then $[\overline{b/\mathtt{o}}]\Delta; [\overline{b/\mathtt{o}}]\Gamma; \overline{\mathtt{X}} \vdash [\overline{b/\mathtt{o}}]e : [\overline{b/\mathtt{o}}]T$

We prove lemmas for the substitution of types for type variables, values for expressions variables, and boxes for expression variables; we prove a preservation lemma for each judgement for each kind of substitution. The substitution lemmas for types and values are standard. The lemmas for boxes are more interesting because the variables being replaced do not have bounds but are subject to constraints. Our lemma therefore requires that the constraints associated with $\overline{\mathtt{o}}$ can be satisfied, rather than the bounds of $\overline{\mathtt{o}}$.

Lemma: closing preserves well-formedness *For all Δ, Γ, $\overline{\mathtt{X}}$, T, $\overline{\mathtt{o}}$, and $\overline{\mathcal{C}}$, if $\Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash T$ OK and $\Delta, \overline{\mathcal{C}}; \Gamma, \overline{\mathtt{o}:\top}; \overline{\mathtt{X}} \vdash \overline{\mathcal{C}}$ OK then $\Delta;\Gamma;\overline{\mathtt{X}} \vdash \Downarrow_{\overline{\mathtt{o}};\overline{\mathcal{C}}} T$* OK

Lemma: closing gives subtypes *For all Δ, Γ, $\overline{\mathtt{X}}$, T, $\overline{b}$, $\overline{\mathtt{o}}$, and $\overline{\mathcal{C}}$, if $\Delta;\Gamma;\overline{\mathtt{X}} \models [\overline{b/\mathtt{o}}]\overline{\mathcal{C}}$ then $\Delta;\Gamma;\overline{\mathtt{X}} \vdash [\overline{b/\mathtt{o}}]T <: \Downarrow_{\overline{\mathtt{o}};\overline{\mathcal{C}}} T$*

In Mojojojo, we introduce existential quantification using the $\Downarrow$ operator. We must account for closing in our proofs and this is done (in part) by the above two lemmas. The first shows that closing preserves the well-formedness of types it operates on. The second shows how closing fits with subtyping: closing and the S-ENV subtyping rule are compliments, they introduce existential types in the same way, this lemma formalises that connection.

5. Towards Effects and Permissions

Mojojojo's ownership types (described in the earlier sections) are purely descriptive. To be useful, an underlying descriptive ownership type system must support an effect system or some prescriptive policy such as owners-as-dominators or owners-as-modifiers. These policies restrict the objects that can be referenced or modified, according to their position in the ownership hierarchy. Although a descriptive system prevents the programmer confusing objects in different contexts (e.g., a value with type `a:C` cannot be stored in a variable with type `b:C`), the benefits usually associated with ownership types are not accorded by a descriptive system. In this section, we explore effects and permissions for Mojojojo; the latter as a generalisation of encapsulation policies.

5.1 An Effect System for Mojojojo

MOJO [9] supported an effect system for multiple ownership; we have designed a similar effect system for Mojojojo. We do not include the formal specification of the effect system in this paper because it is work in progress.

As is standard [11], an effect consists of the region of the heap affected and the kind of effect: read, write, or reference[6]. We use boxes (b, in the formal syntax) to denote the region of the heap affected. Thus, our effects benefit from the same expressivity as the type system. Methods must be annotated with effects by the programmer and are inferred for all expressions. The most interesting aspect of the effect system is that in T-INVK, we must account for unpacked variables in effects. Unpacked variables are treated in the same way as in types, by performing a close operation (using $\Downarrow$). We use the following rule to close boxes (and thus effects), the conclusion should be read as using $\Delta;\Gamma;\overline{\mathtt{X}}$ to judge the closing operationto `b`; the result of closing is `b'`:

$$\frac{\Delta;\Gamma;\overline{\mathtt{X}} \vdash b \subseteq b' \qquad fv(b') \cap \overline{\mathtt{o}} = \emptyset}{\Delta;\Gamma;\overline{\mathtt{X}} \vdash \Downarrow_{\overline{\mathtt{o}};\overline{\mathcal{C}}} b = b'}$$

[6] We have added a reference effect which describes which objects are referenced during execution; this should be useful for aiding memory management.

The close operator finds a $\overline{\mathtt{o}}$-free super-box of b. If b describes an effect, then b' describes a larger effect which does not include unpacked variables. Our effect system is thus safe and conservative in the presence of existential quantification of ownership variables. This rule works in the same way as finding a closed supertype of a type variable in systems with existential types [6].

MOJO's effect system, in contrast, uses wildcard owners directly in effects. The equivalent in Mojojojo would be to allow existential quantification in effects. We found that this approach did not improve expressivity, but did complicate the formal system.

As in MOJO, we can predict the disjointness of expressions if their effects do not overlap. This information could then be used to automatically parallelise expressions [4].

5.2 Policies

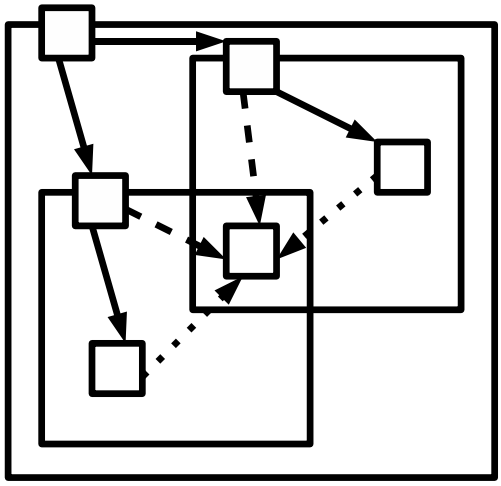

Figure 10. Links in owners-as- policies.

The most common encapsulation policies (owners-as-dominators and owners-as-modifiers) do not scale straightforwardly to multiple ownership. These policies ensure that access to (permission to reference or write) an object is restricted to objects which (transitively) own or are owned by that object. The key question in extending these policies to multiple ownership is how shared ownership is handled. We must decide what makes an 'owner' in terms of owners-as- policies. There are two choices: either an object must have exclusive ownership over an object (by being the only direct owner, or transitively owning all owners, the solid lines in Fig. 10) or shared ownership is good enough (the dashed lines in Fig. 10). We need to allow access to the object in the intersection from somewhere, and should probably forbid access along the dotted lines.

We must also consider the arguments from two sides: is the policy strict enough to be useful, and is it permissive enough to be useful. Requiring exclusive ownership means that an object cannot access objects it partially owns, and, since a common pattern in our examples (see Sect. 3.2) is for an object to have partial ownership over its fields shared with an unknown (existential) owner, this may well be too restrictive. Allowing shared ownership, however, means that the compiler cannot easily reason about access because there may be other (unknown) objects with access. In short, there is no easy answer; we have experimented with systems which require different levels of exclusivity of ownership depending on the direction of reference (inward v. outward), and these appear promising. However, we delay these design decisions and, in Mojojojo, we are

investigating a more precise system of permissions with the aim of encoding any of these policies.

5.3 Permissions

Permissions are, in a sense, the dual of effects: an effect describes what has been done, and a permission describes what may be done. An achievement of Mojojojo is that we use the same language for both permissions and effects; however, the fit is not perfect. The major difference is that with permissions, we care who is doing the action (e.g., *a* may be allowed to modify *b*, but *c* may not be), whereas the subject is irrelevant for effects. In terms of checking, effect and permission information provide no real overlap.

It would be safe to use a method's effect to conservatively estimate if the method can be called for a given receiver and parameters. However, the estimate is too conservative to be useful — the effect of a method may be due to any object that can be named in the method, so it is possible that a method is safe to call even if it has an effect that the receiver or parameters do not have permission for. Instead, we use method-level constraints; by using the method-scoped constraints to check a method, it is always safe to call that method if we can satisfy its constraints at its call site. This system is safe and much more precise than using effects, at the expense of requiring methods to be annotated with both effects and permissions.

Permissions are added as just another kind of constraint; constraints in Mojojojo are therefore not just topological, but describe all kinds of constraint on a program. By encapsulating permissions within constraints, different kinds of permissions can be included without changing the majority of the type system. Permissions on a class allow boxes of objects to read, write, or reference instantiations of that class. Permissions may also be given per-method, these are permissions which have to be satisfied at the call site. Permission constraints may also be present in existential types, in which case they must be satisfied by the hidden witness contexts.

We define the syntax of permissions as:

$$
\begin{array}{lcll}
p & ::= & \texttt{rd} \mid \texttt{wr} \mid \texttt{rf} & \textit{permission kinds} \\
vb & ::= & b \mid b^{\triangledown} \mid b^{\triangle} & \textit{variant boxes} \\
\mathcal{P} & ::= & vb\ p\ b & \textit{permissions} \\
 & & & \\
\mathcal{C} & ::= & \ldots \mid \mathcal{P} & \textit{constraints}
\end{array}
$$

For example, `a rd b` means that all objects in box `a` may read object `b`. The use of variant boxes in permissions allows the programmer to specify that not only a specific object, but any object in a sub- or super-box (using $b^{\triangledown}$or $b^{\triangle}$, respectively) has a permission; for example, $\texttt{a}^{\triangledown}$ includes `a`, $\texttt{a} \cap \texttt{b}$, and $\texttt{a} \cap \texttt{c}$, and $\texttt{a}^{\triangle}$ includes $\texttt{a} \cup \texttt{b}$ and the owner of `a`.

As opposed to other common encapsulation policies, permissions do not propagate up or down the ownership graph. For example, just because object `a` has permission to access object `b`, objects owned by `a` do not automatically have permission to access `b`. The programmer can always add more permissions, or a pre-processing step could enforce a policy by adding propagated permissions.

A key difference between effects and permissions is that effects support a form of subsumption, but permissions do not. We could adopt such a scheme for permissions; however, it would limit the encapsulation policies we could encode (we could support owners-as-modifiers, where permission to modify an object implies permission to modify its owners, but not owners-as-dominators, where permission to reference an object implies permission to reference the objects it owns).

Checking of permissions in the type rules is done implicitly, by checking that class and method level constraints are satisfied in T-NEW (via F-CLASS) and T-INVK, respectively. Our constraint satisfaction rules must expand to accommodate checking of permissions, including accounting for variant boxes, and rules to allow an object complete access to itself and its owner. At runtime, our semantics must keep track of the current object (`this`) in order to type check expressions at intermediate reduction steps; this motivates our use of large-step semantics.

6. Discussion and Future Work

In this paper, we have described and formalised an expressive *core* language; however, this is a long way from a practically usable programming system. Mojojojo's ownership and permission annotations are unwieldy: even simple programming tasks will require complicated types and constraints. We expect that this can be alleviated by choosing sensible defaults, and administering a dose of syntactic sugar. Alternatively, a compiler could infer ownership information and/or effects, using Mojojojo as its internal representation, where there is no requirement for concise and readable code.

Existential types in particular are awkward to write and can be replaced either with variance annotations [18] or a wildcards-like syntax [22]. Topological constraints could be described using keywords, such as 'intersects', 'disjoint', and 'inside', as in MOJO and other ownership systems [9, 12] rather than their underlying set theoretic expressions. Permissions have the most scope for eliminating verbosity, because many fiddly permissions can be described by a single policy definition which can apply across many classes and/or boxes.

We expect that Mojojojo's full expressiveness will only be required in particular parts of some applications: in most cases, carefully chosen defaults and inference should suffice. In particular, we believe that reusable policies (grouping together topological constraints and permissions) have great potential to reduce syntactic overhead. Owners-as-modifiers, to take one example, is a policy where an object "can only be read or written by a partial, transitive owner" [26, 14]. Policies will need to be tailored to capture common usage, such as enforcing design-level encapsulation constraints, or managing memory use. The exact formulation of policies is out of the scope of this paper: a key advantage of Mojojojo is that it can support more than one ownership policy. Mojojojo programs can even use different policies in different places, simply by configuring permissions to implement those policies.

Future work The work presented in this paper forms a complete descriptive ownership system. It is also a work-in-progress snapshot of a system on its way to supporting not only flexible topologies, but also flexible modes of encapsulation in a similarly general manner. To achieve this goal, we require progress in the effects and permissions systems. The former is mostly complete; only the proofs remain (and we do not anticipate any major hurdles there). More work is required on permissions: here, we have the skeleton of a system, but must ensure that it is safe and useful. Safety is to be established by proof, and we expect that changes to our system may be necessary to achieve this. To make permissions useful they must be flexible and reflect actual programming practice. We are confident our permissions are flexible, but currently they are too fine-grained to be useful to a programmer. As described above, we wish to present a programmer-directed abstraction layer on top of our permission system, perhaps reflecting one of the existing ownership-based encapsulation policies. To show that our permissions system does indeed match with programming practice we will perform empirical studies.

7. Related Work

For we are kindred spirits whose powers spring from the same source.

We have described the fundamental ownership concepts in Sect. 2. In this section we describe some systems that seek to achieve the same goals as ours: more flexible ownership and more precise encapsulation properties.

Ownership Domains [2] make ownership hierarchies more flexible by allowing multiple boxes (in Mojojojo terms) per owner. This is the complement of multiple ownership, which allows multiple owners per box. Both approaches have advantages and the systems could easily be combined to make an even more flexible ownership topology[7].

The encapsulation property of Ownership Domains is *link soundness*, which is a generalisation of owners-as-dominators. Links between domains are declared by the programmer, and references may only follow these links. Our permissions system is a generalisation of these links as we extend them to reading and writing as well references.

An alternative to more flexible ownership systems is to drop the notion of ownership altogether and focus on the object invariants themselves. There is a large body of work on object invariants, Summers et. al. [31] surveys much of it and gives examples of object graphs that cannot be described using ownership hierarchies.

DPJ [4] divides the heap into a topology of regions. Regions may overlap and be nested and so topologies of similar expressiveness to our own can be described. A mixture of explicit annotations and paths are used to describe the topology (as opposed to ownership annotations). An expressive language of effects is layered on top of these regions. This can express effects for arrays and subarrays of regions, and for unions of regions, but does not support unknown or partially known regions. DPJ has an 'invokes' effect, but no 'references' effect; we are investigating adding 'invokes' effects and permissions into Mojojojo.

A different approach to parallelisation is proposed by Strok et al. [30]. Here, *access permissions* [3] (similar to fractional permissions, described below, and altogether different from our permissions) are used instead of effects to reason about which statements can be parallelised or re-ordered.

Our permissions are similar to access control lists for file systems. Role-based access control is a similar concept which has been enhanced to be an object-sensitive part of the type system [15]; to the best of our knowledge, this is the closest work to ours, although permissions are given to a role rather than a subset of the heap. Although they share part of their name, fractional permissions [5] and similar systems are very different from ours: they are closer to linear capabilities than they are to access control lists.

8. Conclusion

In this paper, we have presented an ownership type system, Mojojojo, which supports very flexible topologies including multiple owners. Our formalisation of Mojojojo is simple and general. We use standard type theoretic tools (existential quantification, generics) and an elegant system of constraints, based directly on set algebra, to model complex topologies and their variably precise representation in types. Furthermore, we have sketched how effects and permissions can be added to our language; these additions share a common description language. Permissions are incorporated into the existing system of constraints, extending it from a description of a program's runtime topology, to a description of a program's general runtime behaviour.

[7] We have not done so here because the two concepts are orthogonal: there is added complexity, but nothing interesting appears to develop.

Acknowledgments

We would like to thank Sophia Drossopoulou for several useful and detailed discussions about MOJO and Mojojojo. We would like to thank the reviewers for their useful and detailed comments. This work is in part funded by a BuildIT postdoctoral fellowship and a TelstraClear PhD scholarship.

References

[1] Quotes for Mojo Jojo (Character) from "The Powerpuff Girls" (1998). http://www.imdb.com/character/ch0007664/quotes.

[2] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object Oriented Programming (ECOOP)*, 2004.

[3] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[4] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[5] John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.

[6] Nicholas Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, 2009.

[7] Nicholas Cameron and Sophia Drossopoulou. Existential Quantification for Variant Ownership. In *European Symposium on Programming Languages and Systems (ESOP)*, 2009.

[8] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *European Conference on Object Oriented Programming (ECOOP)*, 2008.

[9] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

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

[11] David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[12] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998.

[13] W. R. Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4):305–311, August 1989.

[14] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects (FMCO)*, 2008.

[15] Jeffrey Fischer, Daniel Marino, Rupak Majumdar, and Todd Millstein. Fine-grained access control with object-sensitive roles. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

[16] Aaron Greenhouse and John Boyland. An Object-Oriented Effects System. In *European Conference on Object Oriented Programming (ECOOP)*, 1999.

[17] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus For Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. An earlier version of this work appeared at OOPSLA'99.

[18] Atsushi Igarashi and Mirko Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *Transactions on Programming Languages and Systems*, 28(5):795–847, 2006. An earlier version appeared as "On variance-based subtyping for parametric types" at European Conference on Object Oriented Programming (ECOOP) 2002.

[19] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1988.

[20] Yi Lu, John Potter, and Jingling Xue. Validity Invariants and Effects. In *European Conference on Object Oriented Programming (ECOOP)*, 2007.

[21] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Symposium on Principles of programming languages (POPL)*, 1988.

[22] Mads Torgersen and Erik Ernst and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)*, 2005.

[23] John C. Mitchell and Gordon D. Plotkin. Abstract Types have Existential Type. *Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[24] Nick Mitchell. The Runtime Structure of Object Ownership. In *European Conference on Object Oriented Programming (ECOOP)*, 2006.

[25] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

[26] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.

[27] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *European Conference on Object Oriented Programming (ECOOP)*, 1998.

[28] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[29] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[30] Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Onward! session at Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[31] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming IWACO*, 2009.

A. Elided figures

$$\frac{\mathcal{H} = \overline{\iota \to \{N,\ \overline{\mathtt{f} \to v}\}} \qquad \Delta, \overline{\iota \neq \emptyset}; \Gamma, \overline{\iota \to N}; \overline{\mathtt{X}} \vdash e : T}{\mathcal{H}; \Delta; \Gamma; \overline{\mathtt{X}} \vdash e : T} \text{ (T-RUNTIME)}$$

$$\frac{\mathcal{H} = \overline{\iota \to \{N,\ \overline{\mathtt{f} \to v}\}} \qquad \Delta, \overline{\iota \neq \emptyset}; \Gamma, \overline{\iota \to N}; \overline{\mathtt{X}} \vdash T <: T'}{\mathcal{H}; \Delta; \Gamma; \overline{\mathtt{X}} \vdash T <: T'} \text{ (S-RUNTIME)}$$

$$\frac{\mathcal{H} = \overline{\iota \to \{N,\ \overline{\mathtt{f} \to v}\}} \qquad \Delta, \overline{\iota \neq \emptyset}; \Gamma, \overline{\iota \to N}; \overline{\mathtt{X}} \vdash T \text{ OK}}{\mathcal{H}; \Delta; \Gamma; \overline{\mathtt{X}} \vdash T \text{ OK}} \text{ (F-RUNTIME)}$$

$$\frac{\mathcal{H} = \overline{\iota \to \{N,\ \overline{\mathtt{f} \to v}\}} \qquad \Delta, \overline{\iota \neq \emptyset}; \Gamma, \overline{\iota \to N}; \overline{\mathtt{X}} \models \overline{C}}{\mathcal{H}; \Delta; \Gamma; \overline{\mathtt{X}} \models \overline{C}} \text{ (C-RUNTIME)}$$

Figure 11. Mojojojo runtime rules.

Operational semantics: $\boxed{e; \mathcal{H} \rightsquigarrow e; \mathcal{H}}$

$$\frac{\mathcal{H}(\iota) = \{R; \overline{\mathtt{f} \to v}\}}{\iota.\mathtt{f}_i; \mathcal{H} \rightsquigarrow v_i; \mathcal{H}} \text{ (R-FIELD)}$$

$$\frac{\mathcal{H}(\iota)\ undefined \qquad fields(\mathtt{C}) = \overline{\mathtt{f}} \qquad \mathcal{H}' = \mathcal{H}, \iota \to \{r{:}\mathtt{C}\texttt{<}\overline{T}\texttt{>}; \overline{\mathtt{f} \to \mathtt{null}}\}}{\mathtt{new}\ r{:}\mathtt{C}\texttt{<}\overline{T}\texttt{>}; \mathcal{H} \rightsquigarrow \iota; \mathcal{H}'} \text{ (R-NEW)}$$

$$\frac{\mathcal{H}(\iota) = \{R; \overline{\mathtt{f} \to \mathtt{v}}\} \qquad e; \mathcal{H} \rightsquigarrow v; \mathcal{H}'' \qquad \mathcal{H}' = \mathcal{H}''[\iota \mapsto \{R; \overline{\mathtt{f} \to v}[\mathtt{f}_i \mapsto v]\}]}{\iota.\mathtt{f}_i\ \mathtt{=}\ e; \mathcal{H} \rightsquigarrow v; \mathcal{H}'} \text{ (R-ASSIGN)}$$

$$\frac{\mathcal{H}(\iota) = \{r{:}\mathtt{C}\texttt{<}\overline{T}\texttt{>}; ...\} \qquad e; \mathcal{H} \rightsquigarrow \iota'; \mathcal{H}'' \qquad mBody(\mathtt{m}, r{:}\mathtt{C}\texttt{<}\overline{T}\texttt{>}) = (\mathtt{x}; e') \qquad \mathtt{[}\iota'\mathtt{/x,}\iota\mathtt{/this]}e'; \mathcal{H}'' \rightsquigarrow v; \mathcal{H}'}{\iota.\mathtt{m}(e); \mathcal{H} \rightsquigarrow v; \mathcal{H}';} \text{ (R-INVK)}$$

$$\frac{e; \mathcal{H} \rightsquigarrow \mathtt{null}; \mathcal{H}'}{\iota.\mathtt{m}(e); \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}';} \text{ (R-INVK-NULL)}$$

$$\frac{e; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}'}{\iota.\mathtt{f}_i\ \mathtt{=}\ e; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}'} \text{ (R-ASSIGN-ERR)}$$

$$\frac{e; \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}'}{\iota.\mathtt{m}(e); \mathcal{H} \rightsquigarrow \mathtt{err}; \mathcal{H}';} \text{ (R-INVK-ERR)}$$

Figure 12. Mojojojo reduction rules.